

UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Informática

Fundamentos de Programación con el Lenguaje de Programación C++

Vicente Benjumea y Manuel Roldán

23 de octubre de 2017

Contenido

Prólogo

En este manual se describen las características básicas del lenguaje C++. Está concebido desde el punto de vista docente, por lo que nuestra intención no es hacer una descripción completa del lenguaje, sino únicamente de aquellas características adecuadas como base para facilitar el aprendizaje en un primer curso de programación. Se supone que el alumno compatibilizará el uso de este manual con las explicaciones del profesor, impartidas en el aula.

El lenguaje de programación C++ es un lenguaje muy flexible y versátil. Debido a ello, si se utiliza sin rigor puede dar lugar a construcciones y estructuras de programación complejas, difíciles de comprender y propensas a errores. Por este motivo, restringiremos tanto las estructuras a utilizar como la forma de utilizarlas.

Este manual ha sido elaborado en el Dpto. de Lenguajes y Ciencias de la Computación de la Universidad de Málaga

La última versión de este documento puede ser descargada desde la siguiente página web:

`http://www.lcc.uma.es/%7Evicente/docencia/index.html`

o directamente desde la siguiente dirección:

`http://www.lcc.uma.es/%7Evicente/docencia/cpp/programacion_cxx.pdf`

Parte I

Programación Básica

Capítulo 1

Un Programa C++

En general, un *programa C++* suele estar escrito en diferentes ficheros. Durante el proceso de compilación estos ficheros serán combinados adecuadamente y traducidos a código objeto, obteniendo el programa ejecutable. Nosotros comenzaremos tratando con programas sencillos, para los que bastará un único fichero cuya extensión será una de las siguientes: ".cpp", ".cxx", ".cc", etc. En el capítulo 10 comenzaremos a estudiar cómo estructurar programas complejos en diferentes ficheros.

En este capítulo nos centraremos en presentar los elementos imprescindibles y en mostrar cómo trabajar con el fichero que contiene el programa para generar su correspondiente fichero ejecutable. En posteriores capítulos trataremos con detalle cada uno de los elementos que puede contener un programa.

El fichero suele comenzar con unas líneas para incluir las definiciones de los módulos de biblioteca que utilice nuestro programa, e irá seguido de declaraciones y definiciones de tipos, de constantes y de subprogramas. El programa debe contener un *subprograma* especial (la función *main*) que indica dónde comienza la ejecución. Las instrucciones contenidas en dicha función `main` se ejecutarán una tras otra hasta llegar a su fin. La función *main* devuelve un valor que indica si el programa ha sido ejecutado correctamente o, por el contrario, ha ocurrido un error. En caso de no aparecer explícitamente una sentencia `return`, por defecto, se devolverá un valor que indica terminación normal (0).

A continuación, mostramos un programa que convierte una cantidad determinada de *euros* a su valor en *pesetas* y describimos cómo hay que proceder para obtener el programa ejecutable correspondiente. Más adelante iremos introduciendo cada uno de los elementos que lo forman.

```
//- fichero: euros.cpp --------------------------------------------
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
   // return 0 ;
}
//- fin: euros.cpp ------------------------------------------------
```

Para obtener el programa ejecutable necesitamos dos herramientas básicas: un editor con el que crear un archivo con el texto del mismo (el código fuente) y un compilador para traducir el código fuente a código ejecutable. Para realizar estos pasos podemos seguir dos enfoques diferentes: usar directamente la *línea de comandos* del sistema operativo o usar un *entorno integrado de desarrollo* (IDE).

Si optamos por seguir los pasos desde la línea de comandos (en un entorno linux) podemos usar cualquier editor de texto (emacs, vi, gedit, kate, etc.) para crear un fichero (por ejemplo, `euros.cpp`) y un compilador como *GNU GCC*. En este caso compilaremos de la siguiente forma:

```
g++ -ansi -Wall -Werror -o euros euros.cpp
```

Durante el proceso de compilación pueden aparecer errores, que habrá que solucionar. Cada vez que modifiquemos el fichero para corregir errores deberemos volver a compilar, hasta que no quede ningún error. En ese momento ya tenemos un fichero ejecutable (denominado `euros`) y podemos proceder a su ejecución como se muestra a continuación, donde el texto enmarcado corresponde a una entrada de datos por parte del usuario:

```
Introduce la cantidad (en euros): 3.5 ENTER
3.5 Euros equivalen a 582.351 Pts
```

En un entorno integrado de desarrollo (por ejemplo, Code::Blocks) disponemos de un conjunto de ventanas y de botones asociados a las diferentes herramientas mencionadas. Seguiremos los mismos pasos pero, en lugar de hacerlo desde la línea de comandos, trabajaremos en ventanas propias del entorno y utilizando los botones asociados a las diferentes herramientas indicadas anteriormente.

A continuación, utilizaremos nuestro programa `euros.cpp` para introducir los elementos básicos de un programa. Posteriormente, todos estos elementos serán tratados con mayor amplitud en el capítulo que corresponda.

- *Bibliotecas*

  El lenguaje C++ consta de un reducido número de instrucciones, pero ofrece un amplio repertorio de bibliotecas con herramientas que pueden ser importadas por los programas cuando son necesarias. Por este motivo, un programa suele comenzar por tantas líneas `#include` como bibliotecas se necesiten. Como se puede observar, en nuestro ejemplo se incluye la biblioteca `iostream`, necesaria cuando se van a efectuar operaciones de entrada (lectura de datos) o salida (escritura de datos). Para utilizar la biblioteca `iostream` es necesario utilizar el *espacio de nombres* `std`, éste es un concepto avanzado que, por ahora, está fuera de nuestro ámbito de estudio (se estudiará en el capítulo 10). Por ahora nos basta con recordar que nuestros programas deben contener la siguiente directiva:

  ```
  using namespace std ;
  ```

- *Constantes simbólicas y constantes literales*

  En nuestro ejemplo, sabemos que cada euro equivale a 166.386 pesetas y usamos dicho valor para efectuar operaciones con las que convertir de unas unidades a otras. Podemos emplear dicho valor en su sentido literal (tal y como está escrito), sin embargo, hemos preferido definir un nombre para referirnos al mismo. De esta forma, durante el programa en lugar de referirnos al factor de conversion mediante su valor literal (166.386), podremos usar la nueva constante simbólica `EUR_PTS`. Ello resulta más claro y permitirá que el programa sea más legible y fácil de modificar en el futuro.

  Antes de utilizar una constante simbólica es necesario informar al compilador, indicando de qué tipo es, su nombre (su *identificador*) y el valor que tiene asociado. Si no lo hiciéramos, el compilador no sería capaz de reconocer qué es `EUR_PTS` y nos mostraría un mensaje de error. En nuestro ejemplo utilizamos lo siguiente:

  ```
  const double EUR_PTS = 166.386 ;
  ```

- *Identificadores*

  Para cada elemento que introduzcamos en nuestro programa debemos definir un nombre (identificador) con el que hacer referencia al mismo y disponer de algún mecanismo para informar al compilador de dicho nombre y de sus características. Hemos visto cómo hacerlo para

constantes simbólicas, pero de igual forma habrá que proceder con otro tipo de elementos, como variables, tipos, subprogramas, etc, que iremos tratando en posteriores capítulos.

En C++ se considera que las letras minúsculas son caracteres diferentes de las letras mayúsculas. Como consecuencia, si en nuestro ejemplo hubiéramos usado el identificador `eur_pts` el compilador consideraría que no se trata de `EUR_PTS`, informando de un mensaje de error al no poder reconocerlo. Pruebe, por ejemplo, a sustituir en la siguiente línea el identificador `EUR_PTS` por `eur_pts` y vuelva a compilar el programa. Podrá comprobar que el compilador informa del error con un mensaje adecuado al mismo.

```
double pesetas = euros * EUR_PTS ;
```

Un identificador debe estar formado por una secuencia de letras y dígitos en la que el primer carácter debe ser una letra. El carácter `'_'` se considera como una letra, sin embargo, los nombres que comienzan con dicho carácter se reservan para situaciones especiales, por lo que no deberían utilizarse en programas. Aunque el lenguaje no impone ninguna restricción adicional, en este manual seguiremos unos *criterios de estilo* a la hora decidir qué identificador utilizamos para cada elemento. Ello contribuye a mejorar la legibilidad del programa. Por ejemplo, los identificadores utilizados como nombres de constantes simbólicas estarán formados por letras mayúsculas y, en caso de querer que tengan más de una palabra, usaremos `'_'` como carácter de separación (`EUR_PTS`). En posteriores capítulos iremos proporcionando otros criterios de estilo para ayudar a construir identificadores para variables, tipos, funciones, etc.

- *Palabras reservadas*

  Algunas palabras tienen un significado especial en el lenguaje y no pueden ser utilizadas con otro sentido. Por este motivo no pueden ser utilizados para designar elementos definidos por el programador. Son palabras reservadas, como por ejemplo: `using`, `namespace`, `const`, `double`, `int`, `char`, `bool`, `void`, `for`, `while`, `do`, `if`, `switch`, `case`, `default`, `return`, `typedef`, `enum`, `struct`, etc.

- *Delimitadores*

  Son símbolos que indican comienzo o fin de una entidad (`( ) { } ; , < >`). Por ejemplo, en nuestro programa `euros.cpp` usamos `{` y `}` para delimitar el comienzo y el final de la función `main`, y el símbolo `;` para delimitar el final de una sentencia.

- *Operadores*

  Son símbolos con significado propio según el contexto en el que se utilicen. Ejemplo: `= << >> * / % + - < > <= >= == != ++ -- . ,` etc.

- *Comentarios y formato del programa*

  El compilador necesita reconocer los diferentes elementos que forman el programa. Para ello, utiliza delimitadores y caracteres que permiten separar unos elementos con otros. Además, el programador puede añadir caracteres como espacios en blanco, líneas en blanco o tabuladores para mejorar la legibilidad del programa. El compilador los ignora, salvo que les sirva como separadores de unos elementos y otros.

  Además, el programador puede estar interesado en mejorar la legibilidad del programa incluyendo comentarios dirigidos a otros programadores, no al compilador. Para ello, debe marcar dicho texto de alguna forma que permita ser identificado por el compilador. En C++ esto se puede hacer de dos formas diferentes:

  - Comentarios hasta fin de línea: los símbolos `//` marcan el comienzo del comentario, que se extiende hasta el final de la línea.

    ```
    //- fichero: euros.cpp --------------------------------------------
    ```

- Comentarios enmarcados: los símbolos `/*` marcan el comienzo del comentario, que se extiende hasta los símbolos del fin del comentario `*/`. Por ejemplo, podríamos haber incluido al principio del fichero `euros.cpp` algunas líneas de comentario informando de su autor y fecha de elaboración.

```
/*
 * Autor: Juan Gil
 * Fecha: 12/09/2012
 */
```

- *Variables*

  Los datos se almacenan en memoria en variables de un cierto *tipo*. El programador debe decidir qué variables va a utilizar, pensar en un identificador para referirse a ellas y comunicarlo al compilador. En nuestro programa usamos una variable llamada `euros` para almacenar la cantidad de euros a convertir a pesetas. Usamos la siguiente definición para informar al compilador que `euros` es una variable que contiene un número real:

```
double euros ;
```

- *Entrada/Salida de datos*

  En general, un programa necesitará tomar datos de entrada y mostrar resultados en algún dispositivo de salida. En C++ disponemos de flujos de entrada (`cin`) y de salida (`cout`), que nos permiten efectuar entrada y salida de datos, respectivamente. El operador `>>` se usa para tomar un valor de la entrada y el operador `<<` se usa para sacar un valor por la salida. En nuestro ejemplo, la siguiente sentencia muestra por pantalla un mensaje en el que se solicita que el usuario introduzca un número que representa los euros que se desean convertir a pesetas:

```
cout << "Introduce la cantidad (en euros): " ;
```

  El texto entre comillas es una constante literal, que aparecerá tal y como está escrito en el dispositivo de salida. Si en lugar de una constante literal hubiéramos usado un identificador con el nombre de una constante o de una variable, entonces el valor resultante en la salida sería aquél asociado a dicho identificador. En la siguiente línea sacamos por la salida una combinación de valores almacenados en variables y de constantes literales (entre comillas):

```
cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
```

  En nuestro ejemplo necesitamos que el usuario introduzca como entrada un número que representa los euros a convertir en pesetas. El número introducido debe ser almacenado en alguna variable para poder ser posteriormente manipulado en el programa. Por ese motivo, informamos de la variable que vamos a utilizar (`euros`). En la siguiente sentencia utilizamos el operador `>>` para tomar un número del flujo de entrada (`cin`) y almacenarlo en dicha variable.

```
cin >> euros ;
```

- *Asignación*

  La conversión de euros a pesetas se realiza mediante un cálculo simple, en el que se utiliza el factor de conversión. En nuestro programa queremos realizar el cálculo y almacenar el resultado de manera que pueda ser utilizado posteriormente. Para ello utilizamos una instrución de asignación, en la que el operador = separa el valor a almacenar, resultado de evaluar una expresión aritmética (a la derecha), de la variable en la que lo queremos almacenar (*asignar*).

```
double pesetas = euros * EUR_PTS ;
```

Capítulo 2

Tipos Simples

Un programa trabaja con datos con unas determinadas características. No es lo mismo procesar números naturales, números reales o nombres (cadenas de caracteres). En cada caso tratamos con datos de *tipo* diferente. C++ dispone de un número reducido de tipos predefinidos y es importante elegir en cada caso aquel que mejor se adapta a los datos que deseamos manipular. Por ejemplo, en nuestro programa `euros.cpp` hicimos uso del tipo `double` porque nuestro objetivo era manipular datos numéricos, posiblemente con decimales. También es posible definir nuevos tipos, con características no predefinidas sino pensadas específicamente para un determinado programa.

La elección de un determinado tipo u otro para una determinada entidad del programa proporciona información acerca de:

- El rango de posibles valores que puede tomar.
- El conjunto de operaciones y manipulaciones aplicables.
- El espacio de almacenamiento necesario para almacenar dichos valores.
- La interpretación del valor almacenado.

Por ejemplo, al especificar que un dato es de tipo `double` estamos indicando que se trata de un número real, representado internamente en punto flotante con doble precisión (normalmente usando 8 bytes) y que se puede manipular con los operadores predefinidos `+`, `-`, `*` y `/`.

Los tipos se pueden clasificar en simples y compuestos. Los *tipos simples* se caracterizan porque sus valores son indivisibles, es decir, no se dispone de operadores para acceder a parte de ellos. Por ejemplo, si 126.48 es un valor de tipo `double`, podemos usar el operador `+` para sumarlo con otro, pero no disponemos de operadores predefinidos para acceder a sus dígitos individualmente (aunque esto se pueda realizar indirectamente mediante operaciones de bits o aritméticas). Por el contrario, los *tipos compuestos* (ver el capítulo 6) están formados como un agregado o composición de otros tipos, ya sean simples o compuestos.

2.1. Declaración Vs. Definición

Sabemos que el compilador necesita ser informado de cada identificador que sea utilizado en el programa. Para ello se usa una *declaración*, con la que se "presenta" dicho identificador, haciendo referencia a una determinada entidad que debe ser definida en otro lugar del programa.

Con una *definición* se "establecen las características" de una determinada entidad, asociada a un determinado identificador. Toda definición es a su vez también una declaración.

Es obligatorio que por cada entidad sólo exista **una única definición**, aunque pueden existir varias declaraciones. Así mismo, también es obligatorio la declaración de las entidades que se manipulen en el programa, especificando su tipo, identificador, valores, etc. antes de que sean utilizados.

Por ejemplo, el programa `euros.cpp` contiene la definición de la constante `EUR_PTS` y de las variables `euros` y `pesetas`. Dichas definiciones son, a su vez, declaraciones con las que se presentan dichos identificadores al compilador.

2.2. Tipos Simples Predefinidos

El lenguaje de programación C++ proporciona los siguientes *tipos simples predefinidos*:

```
int float double bool char
```

El tipo `int` se utiliza para trabajar con números *enteros*. Su representación suele coincidir con la definida por el tamaño de palabra del procesador sobre el que va a ser ejecutado, hoy día suele ser de 4 bytes (32 bits), aunque en determinados ordenadores puede ser de 8 bytes (64 bits).

> Puede ser modificado para representar un rango de valores menor, mediante el modificador `short` (normalmente 2 bytes [16 bits]) o para representar un rango de valores mayor, mediante el modificador `long` (normalmente 4 bytes [32 bits] u 8 bytes [64 bits]) y `long long` (normalmente 8 bytes [64 bits]). También puede ser modificado para representar solamente números *naturales* (enteros positivos) utilizando el modificador `unsigned`.

Tanto el tipo `float` como el `double` se utilizan para representar números reales en formato de punto flotante. Se diferencian en el rango de valores que se utiliza para su representación interna. El tipo `double` ("doble precisión") se suele representar utilizando 8 bytes ([64 bits]), mientras que el tipo `float` ("simple precisión") se suele representar utilizando 4 bytes [32 bits]). El tipo `double` también puede ser modificado con `long` para representar "cuádruple precisión" (normalmente 12 bytes [96 bits]).

El tipo `bool` se utiliza para representar valores lógicos (o booleanos), es decir, los valores "Verdadero" o "Falso" o las constantes lógicas `true` y `false`. Suele almacenarse en el tamaño de palabra más pequeño posible direccionable (normalmente 1 byte).

El tipo `char` se utiliza para representar caracteres, es decir, símbolos alfanuméricos (dígitos y letras mayúsculas y minúsculas), de puntuación, espacios, control, etc. Normalmente utiliza un espacio de almacenamiento de 1 byte (8 bits) y puede representar 256 valores diferentes (véase 2.10).

Todos los tipos simples tienen la propiedad de ser indivisibles y además, mantener una relación de orden entre sus elementos (se les pueden aplicar los operadores relacionales, véase 2.5). Se les conoce también como tipos ***escalares***. Todos ellos, salvo los de punto flotante (`float` y `double`), tienen también la propiedad de que cada posible valor tiene un único antecesor y un único sucesor. A éstos se les conoce como tipos ***ordinales*** (en terminología C++, también se les conoce como tipos integrales, o enteros). La siguiente tabla muestra las características fundamentales de los tipos simples predefinidos.

Tipo	Bytes	Bits	Min.Valor	Max.Valor
`bool`	1	8	`false`	`true`
`char`	1	8	-128	127
`short`	2	16	-32768	32767
`int`	4	32	-2147483648	2147483647
`long`	4	32	-2147483648	2147483647
`long long`	8	64	-9223372036854775808	9223372036854775807
`unsigned char`	1	8	0	255
`unsigned short`	2	16	0	65535
`unsigned`	4	32	0	4294967295
`unsigned long`	4	32	0	4294967295
`unsigned long long`	8	64	0	18446744073709551615
`float`	4	32	1.17549435e-38	3.40282347e+38
`double`	8	64	2.2250738585072014e-308	1.7976931348623157e+308
`long double`	12	96	3.36210314311209350626e-4932	1.18973149535723176502e+4932

2.3. Tipos Simples Enumerados

En ocasiones el programador puede estar interesado en trabajar con tipos simples cuyos valores no coinciden con las características de ninguno de los tipos simples predefinidos. Entonces puede definir un nuevo *tipo enumerado*, cuyos valores serán aquellos que explícitamente se enumeren. De esta forma se consigue disponer tipos que expresan mejor las características de las entidades manipuladas por el programa, por lo que el programa será más legible y fácil de entender.

Por ejemplo, si en un programa quisiéramos tratar con colores, con lo visto hasta ahora, para hacer referencia a un determinado color habría que seleccionar un tipo predefinido (por ejemplo, `char`) y suponer que cada color será representado por un cierto carácter (por ejemplo, 'A' para el color azul). En su lugar, podemos definir un nuevo tipo `Color` que contenga `AZUL` como uno de sus valores predefinidos. Para ello, haríamos lo siguiente:

```
enum Color {
    ROJO,
    AZUL,
    AMARILLO
} ;
```

En realidad, internamente cada valor de la enumeración se corresponde con un valor natural, siendo el primero el cero e incrementándose de uno en uno. En nuestro ejemplo `ROJO` se corresponde internamente con 0, `AZUL` con 1, etc.

Los tipos enumerados, al ser tipos definidos por el programador, no tiene entrada ni salida predefinida por el lenguaje, sino que deberá ser el programador el que especifique (programe) cómo se realizará la entrada y salida de datos en caso de ser necesaria.

2.4. Constantes y Variables

Podemos dividir las entidades que nuestro programa manipula en dos clases fundamentales: aquellas cuyo valor no varía durante la ejecución del programa (*constantes*) y aquellas otros cuyo valor puede ir cambiando durante la ejecución del programa (*variables*).

Constantes

A su vez, las constantes pueden aparecer como *constantes literales*, aquellas cuyo valor aparece directamente en el programa, y como *constantes simbólicas*, aquellas cuyo valor se asocia a un identificador, a través del cual se representa.

Ejemplos de constantes literales:

- Constantes lógicas (`bool`):

  ```
  false, true
  ```

- Constantes carácter (`char`), el símbolo constante aparece entre comillas simples:

  ```
  'a', 'b', ..., 'z',
  'A', 'B', ..., 'Z',
  '0', '1', ..., '9',
  ' ', '.', ',', ':', ';', ...
  ```

  Así mismo, ciertos caracteres constantes tienen un significado especial (caracteres de escape):

  - `'\n'`: fin de línea (newline)
  - `'\r'`: retorno de carro (carriage-return)
  - `'\b'`: retroceso (backspace)
  - `'\t'`: tabulador horizontal
  - `'\v'`: tabulador vertical

- `'\f'`: avance de página (form-feed)
- `'\a'`: sonido (audible-bell)
- `'\0'`: fin de cadena
- `'\137'`, `'\x5F'`: carácter correspondiente al valor especificado en notación octal y hexadecimal respectivamente

- Constantes cadenas de caracteres literales, la secuencia de caracteres aparece entre comillas dobles (puede contener caracteres de escape):

```
"Hola Pepe"
"Hola\nJuan\n"
"Hola " "María"
```

- Constantes enteras, pueden ser expresadas en decimal (base 10), hexadecimal (base 16) y octal (base 8). El sufijo `L` se utiliza para especificar `long`, el sufijo `LL` se utiliza para especificar `long long`, el sufijo `U` se utiliza para especificar `unsigned`, el sufijo `UL` especifica `unsigned long`, y el sufijo `ULL` especifica `unsigned long long`:

```
123, -1520, 2345U, 30000L, 50000UL, 0x10B3FC23 (hexadecimal), 0751 (octal)
```

- Constantes reales, números en punto flotante. El sufijo `F` especifica `float`, y el sufijo `L` especifica `long double`:

```
3.1415, -1e12, 5.3456e-5, 2.54e-1F, 3.25e200L
```

dónde la notación `-1e12` representa el número real -1×10^12 y `5.3456e-5` representa el número real 5×10^{-5}.

Constantes Simbólicas

Para declarar una constante simbólica se usa la palabra reservada `const`, seguida por su tipo, el nombre simbólico (o identificador) con el que nos referiremos a ella y el valor asociado tras el símbolo (`=`). Usualmente se definen al principio del programa, después de la inclusión de las cabeceras de las bibliotecas. Ejemplos de constantes simbólicas:

```
const bool OK = true ;
const char SONIDO = '\a' ;
const short ELEMENTO = 1000 ;
const unsigned SEGMIN = 60 ;
const unsigned MINHOR = 60 ;
const unsigned SEGHOR = SEGMIN * MINHOR ;
const long ULTIMO = 100000L ;
const long long TAMANO = 1000000LL ;
const unsigned short VALOR = 100U ;
const unsigned FILAS = 200U ;
const unsigned long COLUMNAS = 200UL ;
const unsigned long long NELMS = 2000ULL ;
const float N_E = 2.7182F ;
const double LOG10E = log(N_E) ;
const long double N_PI = 3.141592L ;
const Color COLOR_DEFECTO = ROJO ;
```

Variables

Las *variables* se definen mediante sentencias en las que se especifica el tipo y una lista de identificadores separados por comas. Todos los identificadores de la lista serán variables de dicho tipo. En dicha sentencia es posible asignar un valor inicial a una variable. Para ello, usamos el símbolo = seguido de una expresión con la que se calcula el valor inicial de la variable. En caso de que no se asigne ningún valor inicial, la variable tendrá un valor **inespecificado**.

El valor de una variable podrá cambiar utilizando sentencia de asignación (véase 2.6) o mediante una sentencia de entrada de datos (véase 3.3).

```
char letra ; // valor inicial inespecificado
int edad = 10, contador = 0 ;
double suma, total = 5.0 ;
```

2.5. Expresiones

Un programa se basa en la realización de una serie de cálculos con los que se producen los resultados deseados. Dichos resultados se almacenan en variables y a su vez son utilizados para nuevos cálculos, hasta obtener el resultado final. Los cálculos se producen mediante la evaluación de expresiones en las que se mezclan operadores con operandos (constantes literales, constantes simbólicas o variables).

En caso de que en una misma expresión se utilice más de un operador habrá que conocer la precedencia de cada operador (aplicando el de mayor precedencia), y en caso de que haya varios operadores de igual precedencia, habrá que conocer su asociatividad (si se aplican de izquierda a derecha o de derecha a izquierda). A continuacción mostramos los operadores disponibles (ordenados de mayor a menor precedencia), aunque por ahora únicamente utilizaremos los más simples e intuitivos:

<table>
<tr><th>Operador</th><th>Tipo de Operador</th><th>Asociatividad</th></tr>
<tr><td><code>[] -> .</code></td><td>Binarios</td><td>Izq. a Dch.</td></tr>
<tr><td><code>! ~ - *</code></td><td>Unarios</td><td>Dch. a Izq.</td></tr>
<tr><td><code>* / %</code></td><td>Binarios</td><td>Izq. a Dch.</td></tr>
<tr><td><code>+ -</code></td><td>Binarios</td><td>Izq. a Dch.</td></tr>
<tr><td><code><< >></code></td><td>Binarios</td><td>Izq. a Dch.</td></tr>
<tr><td><code>< <= > >=</code></td><td>Binarios</td><td>Izq. a Dch.</td></tr>
<tr><td><code>== !=</code></td><td>Binarios</td><td>Izq. a Dch.</td></tr>
<tr><td><code>&</code></td><td>Binario</td><td>Izq. a Dch.</td></tr>
<tr><td><code>^</code></td><td>Binario</td><td>Izq. a Dch.</td></tr>
<tr><td><code>|</code></td><td>Binario</td><td>Izq. a Dch.</td></tr>
<tr><td><code>&&</code></td><td>Binario</td><td>Izq. a Dch.</td></tr>
<tr><td><code>||</code></td><td>Binario</td><td>Izq. a Dch.</td></tr>
<tr><td><code>?:</code></td><td>Ternario</td><td>Dch. a Izq.</td></tr>
</table>

Significado de los operadores:

- Aritméticos. El resultado es del mismo tipo que los operandos (véase 2.8):

`- valor`	Menos unario
`valor * valor`	Producto (multiplicación)
`valor / valor`	División (entera o real según el tipo de operandos)
`valor % valor`	Módulo (resto de la división) (sólo tipos enteros)
`valor + valor`	Suma
`valor - valor`	Resta

- Relacionales/Comparaciones. El resultado es de tipo `bool`

`valor < valor`	Comparación menor
`valor <= valor`	Comparación menor o igual
`valor > valor`	Comparación mayor
`valor >= valor`	Comparación mayor o igual
`valor == valor`	Comparación de igualdad
`valor != valor`	Comparación de desigualdad

- Operadores de bits, sólo aplicable a operandos de tipos enteros. El resultado es del mismo tipo que los operandos:

<table>
<tr><td><code>~ valor</code></td><td>Negación de bits (complemento)</td></tr>
<tr><td><code>valor << despl</code></td><td>Desplazamiento de bits a la izq.</td></tr>
<tr><td><code>valor >> despl</code></td><td>Desplazamiento de bits a la dch.</td></tr>
<tr><td><code>valor & valor</code></td><td>AND de bits</td></tr>
<tr><td><code>valor ^ valor</code></td><td>XOR de bits</td></tr>
<tr><td><code>valor | valor</code></td><td>OR de bits</td></tr>
</table>

- Condicional. El resultado es del mismo tipo que los operandos:

`cond ? valor1 : valor2`	Si `cond` es `true` entonces el resultado es `valor1`, en otro caso es `valor2`

- Lógicos, sólo aplicable a operandos de tipo booleano. Tanto el operador `&&` como el operador `||` se evalúan en *cortocircuito*. El resultado es de tipo `bool`:

<table>
<tr><td><code>! valor</code></td><td>Negación lógica</td><td>(Si <code>valor</code> es <code>true</code> entonces <code>false</code>, en otro caso <code>true</code>)</td></tr>
<tr><td><code>valor1 && valor2</code></td><td>AND lógico</td><td>(Si <code>valor1</code> es <code>false</code> entonces <code>false</code>, en otro caso <code>valor2</code>)</td></tr>
<tr><td><code>valor1 || valor2</code></td><td>OR lógico</td><td>(Si <code>valor1</code> es <code>true</code> entonces <code>true</code>, en otro caso <code>valor2</code>)</td></tr>
</table>

  - La tabla de verdad de los operadores lógicos:

x	! x
F	T
T	F

<table>
<tr><th>x</th><th>y</th><th>x && y</th><th>x || y</th></tr>
<tr><td>F</td><td>F</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>T</td><td>F</td><td>T</td></tr>
<tr><td>T</td><td>F</td><td>F</td><td>T</td></tr>
<tr><td>T</td><td>T</td><td>T</td><td>T</td></tr>
</table>

  - Los operadores `&&` y `||` se evalúan en *cortocircuito*, que significa que cuando ya se conoce el resultado de la operación lógica trás la evaluación del primer operando, entonces el segundo operando **no** se evalúa.
    - Para el operador `&&`, cuando el primer operando se evalúa a `false`, entonces el resultado de la operación `&&` es `false` sin necesidad de evaluar el segundo operando.
    - Para el operador `||`, cuando el primer operando se evalúa a `true`, entonces el resultado de la operación `||` es `true` sin necesidad de evaluar el segundo operando.

    Sin embargo, si trás la evaluación del primer operando no se puede calcular el resultado de la operación lógica, entonces **sí** es necesario evaluar el segundo operando para calcular el resultado de la expresión lógica.
    - Para el operador `&&`, cuando el primer operando se evalúa a `true`, entonces el resultado de la operación `&&` es el resultado de evaluar el segundo operando.
    - Para el operador `||`, cuando el primer operando se evalúa a `false`, entonces el resultado de la operación `||` es el resultado de evaluar el segundo operando.

2.6. Sentencias de Asignación

Una sentencia de asignación permite almacenar en una variable el resultado de evaluar una expresión. La variable a la que se asigna un valor debe aparecer **a la izquierda** de la sentencia, separada de la expresión por el operador de asignación (=). La ejecución de una sentencia de asignación consta de dos pasos: en primer lugar se evalúa la expresión que aparece a la derecha (considerando las reglas de precedencia y asociatividad de los operadores) y posteriormente se guarda el valor resultante en la variable que aparece a la izquierda. El valor que hubiera anteriormente almacenado en dicha variable se perderá, siendo sustituido por el nuevo valor asignado.

```
<variable> = <expresión> ;
```

Por ejemplo:

```
const int MAXIMO = 15 ;
int main()
{
    int cnt ;               // valor inicial inespecificado
    cnt = 30 * MAXIMO + 1 ; // asigna a cnt el valor 451
    cnt = cnt + 10 ;        // cnt pasa ahora a contener el valor 461
}
```

En C++ se dispone de operadores para escribir de forma abreviada determinadas expresiones típicas de asignación. Por ejemplo, es frecuente que surja la necesidad de sumar uno al contenido de cierta variable `v`. Para ello podría ejecutar la siguiente sentencia:

```
v = v + 1 ;
```

aunque también es posible obtener el mismo resultado con `++v;` o `v++;`. Nótese que aunque en realidad ambas sentencias dan el mismo resultado, en realidad no son equivalentes. En este curso usaremos las sentencias de asignación de forma que siempre serán equivalentes. Hay otros operadores abreviados de asignación, que mostramos en la siguiente tabla:

Sentencia	Equivalencia
`++variable;`	`variable = variable + 1;`
`--variable;`	`variable = variable - 1;`
`variable++;`	`variable = variable + 1;`
`variable--;`	`variable = variable - 1;`
`variable += expresion;`	`variable = variable + (expresion);`
`variable -= expresion;`	`variable = variable - (expresion);`
`variable *= expresion;`	`variable = variable * (expresion);`
`variable /= expresion;`	`variable = variable / (expresion);`
`variable %= expresion;`	`variable = variable % (expresion);`
`variable &= expresion;`	`variable = variable & (expresion);`
`variable ^= expresion;`	`variable = variable ^ (expresion);`
`variable |= expresion;`	`variable = variable | (expresion);`
`variable <<= expresion;`	`variable = variable << (expresion);`
`variable >>= expresion;`	`variable = variable >> (expresion);`

2.7. Visibilidad de los identificadores

El compilador necesita saber a qué entidad nos estamos refiriendo cuando utilizamos un determinado identificador en un programa. Para ello, es necesario que "presentemos" (declaremos) cada nuevo identificador, así como conocer las reglas que se usan para determinar en qué parte del programa es visible (utilizable). Ello requiere introducir el concepto de *bloque*. Un bloque es una agrupación de instrucciones delimitadas por `{` y `}`. Si un identificador es declarado dentro de un bloque (como ocurre con las variables), su ámbito de visibilidad va desde el punto de la declaración hasta el final de dicho bloque. Sin embargo, si es declarado fuera de cualquier bloque (como ocurre con las constantes simbólicas), su ámbito de visibilidad abarca desde el punto de su declaración hasta el final de la unidad de compilación (por ahora supondremos que la unidad de compilación coincide con el fichero en el que está escrito el programa).

Como consecuencia de la aplicación de esta regla, debemos efectuar la declaración de la variable antes de su uso en un bloque. Por ejemplo, el siguiente trozo de programa ocasionaría un error por parte del compilador

```
{
    cnt = 0 ;
    int cnt ;
}
```

2.8. Conversiones Automáticas (Implícitas) de Tipos

Es posible que nos interese realizar operaciones en las que se mezclen datos de tipos diferentes. El lenguaje de programación C++ realiza *conversiones de tipo automáticas*, de tal forma que el resultado de la operación sea del tipo más amplio de los implicados en ella. *Siempre que sea posible*, los valores se convierten de tal forma que no se pierda información.

Aunque C++ utiliza reglas mucho más detalladas para efectuar conversiones implícitas de tipos, básicamente se utilizan dos:

- Promoción: en cualquier operación en la que aparezcan dos tipos diferentes se eleva el rango del que lo tiene menor para igualarlo al del mayor. El rango de los tipos de mayor a menor es el siguiente:

  `double, float, long, int, short, char`

- Almacenamiento: En una sentencia de asignación, el resultado final de los cálculos se reconvierte al tipo de la variable al que está siendo asignado.

2.9. Conversiones Explícitas de Tipos

También es posible realizar *conversiones de tipo explícitas* ("*castings*"). Para ello, se escribe el tipo al que queremos convertir y entre paréntesis la expresión cuyo valor queremos convertir. Por ejemplo:

`char x = char(65) ;`	produce el carácter 'A' a partir de su código ASCII (65)
`int x = int('a') ;`	convierte el carácter 'a' a su valor entero (97)
`int x = int(ROJO) ;`	produce el entero 0
`int x = int(AMARILLO) ;`	produce el entero 2
`int x = int(3.7) ;`	produce el entero 3
`double x = double(2) ;`	produce el real (doble precisión) 2.0
`Color x = Color(1) ;`	produce el Color AZUL
`Color x = Color(c+1) ;`	si c es de tipo Color, produce el siguiente valor de la enumeración

El tipo enumerado se convierte automáticamente a entero, sin embargo la conversión inversa no se realiza de forma automática. En tal caso sería necesario efectuar una conversión explícita de tipo. Por ejemplo, para incrementar el valor de una variable del tipo enumerado `Color`, podemos sumar uno a la variable `c` (aquí se hace una conversión implícita), y efectuar una conversión explícita del nuevo valor obtenido (que ahora es de tipo entero) al tipo `Color`.

```
enum Color {
    ROJO, AZUL, AMARILLO
} ;
int main()
{
    Color c = ROJO ;
    c = Color(c + 1) ;
    // ahora c tiene el valor AZUL
}
```

2.10. Tabla ASCII

La tabla ASCII es comúnmente utilizada como base para la representación de los caracteres, donde los números del 0 al 31 se utilizan para representar caracteres de control, y los números del 128 al 255 se utilizan para representar caracteres extendidos.

Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb
0	\0	16	DLE	32	SP	48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	\a	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
8	\b	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	\t	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	\n	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	\v	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	\f	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	\r	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

2.11. Algunas consideraciones respecto a operaciones con números reales

La representación interna de los números reales utiliza un número finito de dígitos decimales. Ello hace que tanto el rango de valores que se pueden almacenar como su precisión están limitados. Además, la representación de números reales en base 2 hace que la representación de determinados números en base 10 sea **inexacta**. Como consecuencia, la realización de operaciones aritméticas en punto flotante puede dar lugar a pérdidas de precisión que ocasionen resultados inesperados. Distintas operaciones, que matemáticamente son equivalentes, pueden ser computacionalmente diferentes. Para evitar que ello ocasione errores difíciles de detectar, **evitaremos** la comparación directa (de igualdad o desigualdad) de números reales. Si nos interesa efectuar la comparación tendremos en cuenta el error admitido en la misma. En este momento aún no hemos presentado todos los elementos necesarios para poder entender cómo y por qué hacer esto, por lo que retomaremos este punto más adelante.

```
#include <iostream>
using namespace std ;
int main()
{
    bool ok = (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0) ;
    cout << "Resultado de (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0): "
         << boolalpha << ok << endl ;
}
```

produce un resultado distinto a lo que cabría esperar:

```
Resultado de (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0): false
```

Capítulo 3

Entrada y Salida de Datos Básica

La entrada y salida de datos permite a un programa recibir información desde el exterior (usualmente el teclado), la cual será transformada mediante un determinado procesamiento, y posteriormente permitirá mostrar al exterior (usualmente la pantalla del monitor) el resultado de la computación.

Para poder realizar entrada y salida de datos básica es necesario incluir la biblioteca `iostream`, que contiene las declaraciones de tipos y operaciones que la realizan. Todas las definiciones y declaraciones de la biblioteca estándar se encuentran bajo el espacio de nombres `std` (ver capítulo 10), por lo que para utilizarlos adecuadamente habrá que utilizar la *directiva* `using` al comienzo del programa.

```
#include <iostream>   // inclusión de la biblioteca de entrada/salida
using namespace std ; // utilización del espacio de nombres de la biblioteca
const double EUR_PTS = 166.386 ;
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
}
```

3.1. El "Buffer" de Entrada y el "Buffer" de Salida

Ningún dato de entrada o de salida en un programa C++ se obtiene o envía directamente del/al hardware, sino a través de unas zonas de memoria intermedia (*"buffers"* de entrada y salida, respectivamente), controlados por el Sistema Operativo e independientes de la ejecución de nuestro programa.

Así, cuando se pulsa alguna tecla, el Sistema Operativo almacena en secuencia las teclas pulsadas en una zona de memoria intermedia: *el "buffer" de entrada*. Cuando un programa realiza una operación de entrada de datos (por ejemplo, `cin >> valor`), accede al "buffer" asociado al flujo de entrada `cin` y obtiene la secuencia de caracteres allí almacenados (si los hubiera) o, en caso de estar vacío, esperará hasta que los haya (hasta que el usuario pulse una serie de teclas seguidas por la tecla *"ENTER"*). Una vez obtenida la secuencia de caracteres entrada por el usuario, se convertirá a un valor del tipo especificado por la operación de entrada, y dicho valor se asignará a la variable especificada.

De igual forma, cuando se va a mostrar alguna información de salida (por ejemplo, `cout << val`), dichos datos no van directamente a la pantalla, sino que se convierten a un formato adecuado para ser impresos (una secuencia de caracteres) y se almacenan en una zona de memoria intermedia denominada *"buffer"de salida*, asociado al flujo de salida `cout`, desde donde el Sistema Operativo tomará la información para ser mostrada por pantalla.

3.2. Salida de Datos

La *salida* de datos permite mostrar información al exterior, y se realiza a través de los *flujos de salida*. El flujo de salida asociado a la *salida estándar* (usualmente la pantalla o terminal de la consola) se denomina `cout`. La salida de datos a pantalla se realiza utilizando el operador `<<` sobre el flujo `cout` especificando el dato cuyo valor se mostrará. Por ejemplo:

```
cout << "Introduce la cantidad (en euros): " ;
```

escribirá en la salida estándar el mensaje correspondiente a la cadena de caracteres especificada. El siguiente ejemplo escribe en la salida estándar el valor de las variables `euros` y `pesetas`, así como un mensaje para interpretarlos adecuadamente. El símbolo `endl` especifica que la sentencia deberá escribir un *fin de línea*, que indica que lo que se muestre a continuación se realizará en una nueva línea.

```
cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
```

Salida de Datos Formateada

Cuando se utiliza en operador de salida `<<` para escribir un dato, el resultado que aparece en la salida se ajusta a un comportamiento por defecto. En ocasiones nos puede interesar controlar la forma en la que se muestran los datos, especificando un determinado formato. Para ello, es necesario incluir la biblioteca estándar `iomanip`. Por ejemplo, en el siguiente programa usamos algunos manipuladores típicos:

```
#include <iostream>
#include <iomanip>
using namespace std ;
int main()
{
    bool x = true ;
    cout << boolalpha << x ; // escribe los booleanos como 'false' o 'true'

    cout << dec << 27 ; // escribe 27 (decimal)
    cout << hex << 27 ; // escribe 1b (hexadecimal)
    cout << oct << 27 ; // escribe 33 (octal)

    cout << setprecision(2) << 4.567 ;       // escribe 4.6
    cout << setw(5) << 234 ;                 // escribe "  234"
    cout << setfill('#') << setw(5) << 234 ; // escribe "##234"
}
```

cuyo comportamiento es el siguiente:

- El manipulador `boolalpha` especifica que los valores lógicos se mostrarán mediante los valores `false` y `true`. Si no se especifica, se muestran los valores `0` y `1` respectivamente.
- Los manipuladores `dec`, `hex`, `oct` especifican que la salida se realizará utilizando el sistema de numeración *decimal*, *hexadecimal* u *octal* respectivamente.
- El manipulador `setprecision(...)` especifica la cantidad de dígitos significativos (precisión) que se mostrará en la salida de números reales.
- El manipulador `setw(...)` especifica la anchura (*width*) que como mínimo ocupará la salida da datos (permite mostrar la información de forma tabulada).
- El manipulador `setfill(...)` especifica el carácter de relleno (*fill*) que se utilizará, en caso de ser necesario, para ocupar toda la anchura del campo de salida (especificada con `setw`). Por defecto, si no se especifica el carácter de relleno, se utiliza el *espacio en blanco* (`' '`).

3.3. Entrada de Datos

La *entrada* de datos permite recibir información desde el exterior, y se realiza a través de los *flujos de entrada*. El flujo de entrada asociado a la *entrada estándar* (usualmente el teclado) se denomina `cin`. La entrada de datos desde el teclado se realiza mediante el operador `>>` sobre el flujo `cin` especificando la variable donde almacenar el valor de entrada leído desde el teclado:

```
cin >> euros ;
```

incluso es posible leer varios valores consecutivamente en la misma sentencia de entrada, de tal forma que las siguientes sentencias son equivalentes:

`cin >> minimo ;` `cin >> maximo ;`	*es equivalente a*	`cin >> minimo >> maximo ;`

El operador de entrada `>>` se comporta de la siguiente forma: elimina los espacios en blanco[1] que hubiera al principio del flujo de entrada de datos, y lee de dicho flujo de entrada una secuencia de caracteres hasta que encuentre algún carácter no válido (según el tipo de la variable que almacenará la entrada de datos), que no será leído y permanecerá disponible en el flujo de entrada de datos hasta que se realice la próxima operación de entrada. La secuencia de caracteres leída del flujo de entrada será manipulada (convertida) para obtener el valor correspondiente del tipo adecuado que será almacenado en la variable especificada.

En caso de que durante la operación de entrada de datos surja alguna situación de error, dicha operación de entrada se detiene y el flujo de entrada se pondrá en un estado erróneo.

Por ejemplo, dado el siguiente programa:

```
#include <iostream>
using namespace std ;
int main()
{
   int num_1, num_2 ;
   cout << "Introduce el primer número: " ;
   cin >> num_1 ;
   cout << "Introduce el segundo número: " ;
   cin >> num_2 ;
   cout << "Multiplicación: " << (num_1 * num_2) << endl ;
   cout << "Fin" << endl ;
}
```

si al ejecutarse, el usuario introduce [12 ENTER] como primer número, y [3 ENTER] como segundo número, se produce la siguiente salida:

```
Introduce el primer número: [12 ENTER]
Introduce el segundo número: [3 ENTER]
Multiplicación: 36
Fin
```

El funcionamiento detallado es el siguiente:

1. La ejecución del programa escribe en pantalla el mensaje para que el usuario introduzca el primer número.
2. Al estar vacío el flujo de entrada, el operador `>>` detiene la ejecución en espera de datos. Cuando el usuario introduce una entrada de datos [12 ENTER], el operador `>>` extrae del flujo de entrada `cin` la secuencia de caracteres `'1'` y `'2'`, deteniendo la extracción al encontrar el carácter `'ENTER'`. Estos caracteres se convierten al número `12` (`1 * 10 + 2`) que será almacenado en la variable `num_1`.

[1] Se consideran espacios en blanco los siguientes caracteres: espacio en blanco (`' '`), tabuladores (`'\t'`, `'\v'` y `'\f'`), retorno de carro (`'\r'`) y nueva línea (`'\n'`).

3. Posteriormente, el programa escribe en pantalla el mensaje para que el usuario introduzca el segundo número.
4. En la siguiente operación de entrada, el operador `>>` elimina el carácter `'ENTER'` del flujo de entrada `cin`, remanente de la operación anterior. Al quedar vacío el flujo de entrada, la operación de entrada se detiene hasta que haya nuevos caracteres disponibles. Cuando el usuario introduce una nueva entrada [3 ENTER], el operador `>>` extrae del flujo de entrada `cin` la secuencia de caracteres `'3'`, deteniendo la extracción al encontrar el carácter `'ENTER'`. Estos caracteres se convierten al número `3` que será almacenado en la variable `num_2`.
5. Finalmente se realiza la multiplicación de ambos números, mostrando el resultado (`36`).

Sin embargo, si al ejecutarse de nuevo el programa, el usuario introduce [12 ESPACIO 3 ENTER] como primer número, entonces la ejecución del programa no se detiene para permitir introducir el segundo número, ya que éste ya se encuentra en el flujo de entrada de datos, por lo que se produce la siguiente salida:

```
Introduce el primer número: [12 [ESP] 3 [ENTER]]
Introduce el segundo número: Multiplicación: 36
Fin
```

El funcionamiento detallado es el siguiente:

1. La ejecución del programa escribe en pantalla el mensaje para que el usuario introduzca el primer número.
2. Al estar vacío el flujo de entrada, el operador `>>` detiene la ejecución en espera de datos. Cuando el usuario introduce una entrada de datos [12 ESPACIO 3 ENTER], el operador `>>` extrae del flujo de entrada `cin` la secuencia de caracteres `'1'` y `'2'`, deteniendo la extracción al encontrar el carácter `'ESPACIO'`. Estos caracteres se convierten al número `12` (`1 * 10 + 2`) que será almacenado en la variable `num_1`.
3. Posteriormente, el programa escribe en pantalla el mensaje para que el usuario introduzca el segundo número.
4. En la siguiente operación de entrada, el operador `>>` elimina el carácter `'ESPACIO'` del flujo de entrada `cin` y encuentra caracteres disponibles en el flujo de entrada [3 ENTER], remanentes de la operación anterior, por lo que el operador `>>` no detendrá la ejecución del programa, y extrae del flujo de entrada `cin` la secuencia de caracteres `'3'`, deteniendo la extracción al encontrar el carácter `'ENTER'`. Estos caracteres se convierten al número `3` que será almacenado en la variable `num_2`.
5. Finalmente se realiza la multiplicación de ambos números, mostrando el resultado (`36`).

Entrada de Datos Avanzada

Hay ocasiones en la que nos interesa obtener caracteres del flujo de entrada sin saltar los espacios en blanco, ya que por razones de procesamiento, nos puede ser útil el procesamiento de dichos caracteres. Por ello, el siguiente operador nos permite leer, desde el flujo de entrada, un único carácter, sin eliminar los espacios iniciales, de tal forma que éstos también puedan ser procesados:

```
{
    char c ;
    cin.get(c) ; // lee un carácter sin eliminar espacios en blanco iniciales
    ...
}
```

En caso de querer eliminar los espacios iniciales explícitamente, el manipulador `ws` realizará dicha operación de eliminación de los espacios iniciales:

```
{
    char c ;
    cin >> ws ;  // elimina los espacios en blanco iniciales
```

```
        cin.get(c) ; // lee sin eliminar espacios en blanco iniciales
        ...
    }
```

Es posible también eliminar un número determinado de caracteres del flujo de entrada, o hasta que se encuentre un determinado carácter:

```
    {
        cin.ignore() ;            // elimina el próximo carácter
        cin.ignore(5) ;           // elimina los 5 próximos caracteres
        cin.ignore(1000, '\n') ; // elimina 1000 caracteres o hasta ENTER (nueva-línea)
    }
```

La entrada y salida de cadenas de caracteres se puede ver en los capítulos correspondientes (cap. 6.2.1).

3.4. Control del Estado del Flujo

Cuando se realiza una entrada de datos errónea, el flujo de entrada se pondrá en un *estado de error*, de tal forma que cualquier operación de entrada de datos sobre un flujo de datos en estado erróneo también fallará. Por ejemplo, la ejecución del siguiente programa entrará en un "bucle sin fin" en caso de que se introduzca una letra en vez de un número, ya que el valor que tome la variable `n` no estará en el rango adecuado, y cualquier otra operación de entrada también fallará, por lo que el valor de `n` nunca podrá tomar un valor válido dentro del rango especificado:

```
int main()
{
    int n = 0;
    do {
        cout << "Introduzca un numero entre 1 y 9: ";
        cin >> n;
    } while (! (n > 0 && n < 10));
    cout << "Valor: " << n << endl;
}
```

Sin embargo, es posible comprobar el estado de un determinado flujo de datos, y en caso de que se encuentre en un estado de error, es posible restaurarlo a un estado correcto, por ejemplo:

```
int main()
{
    int n = 0;
    do {
        cout << "Introduzca un numero 1 y 9: ";
        cin >> n;
        while (cin.fail()) {                    // ¿ Estado Erróneo ?
            cin.clear();                        // Restaurar estado
            cin.ignore(1000, '\n');             // Eliminar la entrada de datos anterior
            cout << "Error: Introduzca un numero 1 y 9: ";
            cin >> n;
        }
    } while (! (n > 0 && n < 10));
    cout << "Valor: " << n << endl;
}
```

Capítulo 4

Estructuras de Control

El lenguaje de programación C++ dispone de estructuras de control muy flexibles. Aunque ello es positivo, un abuso de la flexibilidad proporcionada por el lenguaje puede dar lugar a programas con estructuras complejas y características no deseables. Por ello, sólo veremos algunas de ellas y las utilizaremos en contextos y situaciones restringidas. Todas aquellas estructuras que no se presenten en este texto no serán utilizadas en el curso.

4.1. Sentencia, Secuencia y Bloque

En C++ la unidad fundamental de acción es la sentencia, y expresamos la composición de sentencias como una *secuencia de sentencias* terminadas por el carácter "punto y coma" (`;`). Su orden de ejecución (*flujo de ejecución*) es *secuencial*, es decir, cada sentencia se ejecuta cuando termina la anterior, siguiendo el orden en el que están escritas en el texto del programa.

Un *bloque* es una unidad de ejecución mayor que la sentencia, y permite agrupar una secuencia de sentencias como una unidad. Para ello, formaremos un bloque delimitando entre dos llaves la secuencia de sentencias que agrupa. Es posible anidar bloques, es decir, se pueden definir bloques dentro de otros bloques.

```
int main()
{
    <sentencia_1> ;
    <sentencia_2> ;
    {
        <sentencia_3> ;
        <sentencia_4> ;
        . . .
    }
    <sentencia_n> ;
}
```

4.2. Declaraciones Globales y Locales

En el capítulo 2 tratamos acerca del ámbito de visibilidad de los identificadores y lo aplicamos a nuestro ejemplo básico, haciendo referencia a la declaración de constantes y variables. En este capítulo utilizaremos bloques que, como hemos visto, pueden estar anidados dentro de otros bloques. Ello introduce la posibilidad de declarar identificadores en diferentes bloques, pudiendo entrar en conflicto unos con otros, por lo que debemos conocer las reglas utilizadas en el lenguaje C++ para decidir a qué identificador nos referimos en cada caso. Distinguiremos dos clases de declaraciones: globales y locales.

Entidades globales son aquellas que han sido definidas fuera de cualquier bloque. Su *ámbito de visibilidad* comprende desde el punto en el que se definen hasta el final del fichero. Este tipo

de entidades se crean al principio del programa y se destruyen al finalizar éste. Por ejemplo, la constante simbólica `EUR_PTS`, utilizada en nuestro primer programa, es global y visible (utilizable) desde el punto en que se declaró hasta el final del fichero. En capítulos posteriores estudiaremos otras entidades que se declaran con un ámbito global, como tipos definidos por el programador (ya hemos visto tipos enumerados), prototipos de subprogramas y definiciones de subprogramas. También es posible definir variables con un ámbito global, aunque es una práctica que, en general, está desaconsejada y no seguiremos en este curso.

Entidades locales son aquellas que se definen dentro de un bloque. Su *ámbito de visibilidad* comprende desde el punto en el que se definen hasta el final de dicho bloque. Este tipo de entidades se crean en el punto donde se realiza la definición, y se destruyen al finalizar el bloque. Normalmente serán variables locales, aunque también es posible declarar constantes localmente. Sin embargo, por lo general en este curso seguiremos el criterio de declarar las constantes globalmente escribiendo su definición al principio del programa, fuera de cualquier bloque.

```
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;       // Declaración de constante GLOBAL
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;                      // Declaración de variable LOCAL
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;  // Declaración de variable LOCAL
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
}
```

En caso de tener declaraciones de diferentes entidades con el mismo identificador en diferentes niveles de anidamiento, la entidad visible será aquella que se encuentre declarada en el bloque de nivel de anidamiento más interno. La declaración más interna *oculta* a aquella más externa. Sin embargo, se considera que no es una buena práctica de programación ocultar identificadores al redefinirlos en niveles de anidamiento más internos, ya que ello conduce a programas difíciles de leer y propensos a errores, por lo que procuraremos evitar esta práctica.

En el siguiente ejemplo declaramos una variable local `x` de tipo `int` y la inicializamos con el valor 3. Posteriormente declaramos una nueva variable `x`, diferente de la anterior, de tipo `double` y con valor inicial 5.0. En el bloque interno coexisten ambas variables en memoria, cada una de tipo diferente, pero cualquier uso del identificador `x` hará referencia a la variable de tipo `double`, porque está declarada en un ámbito más interno, ocultando así a la variable `x` de tipo `int`.

```
int main()
{
    int x = 3 ;
    int z = x * 2 ;        // x es vble de tipo int con valor 3
    {
        double x = 5.0 ;
        double n = x * 2 ;  // x es vble de tipo double con valor 5.0
    }
    int y = x + 4 ;        // x es vble de tipo int con valor 3
}
```

4.3. Sentencias de Selección

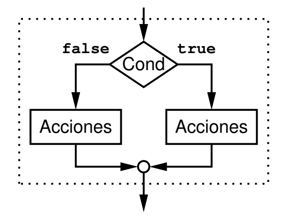
Hasta ahora hemos considerado instrucciones de asignación y de entrada/salida, y el flujo de ejecución es secuencial. Sin embargo, este tipo de instrucciones no basta si necesitamos programas en los que queremos optar entre diferentes comportamientos, dependiendo de diferentes condiciones. Por ejemplo, podríamos estar interesados en disponer de un programa de conversión de monedas que funcione, tanto para convertir de euros a pesetas como al contrario. En este caso, el usuario informaría en primer lugar del tipo de conversión que desea y posteriormente se aplicaría una

conversión u otra. Para poder describir este comportamiento necesitamos sentencias de selección, que permitan efectuar preguntas y seleccionar el comportamiento adecuado en función del resultado de las mismas. Las preguntas serán expresadas mediante *expresiones lógicas*, que devuelven un valor de tipo `bool`. El valor resultante de evaluar la expresión lógica (`true` o `false`), será utilizado para seleccionar el bloque de sentencias a ejecutar, descartando el resto de alternativas posibles.

Sentencia if

La sentencia de selección se puede utilizar con diferentes formatos. Para el ejemplo mencionado usaríamos una sentencia de selección condicional compuesta:

```
if ( <expresión_lógica> ) {
    <secuencia_de_sentencias_v> ;
} else {
    <secuencia_de_sentencias_f> ;
}
```



cuya ejecución comienza evaluando la expresión lógica. Si su resultado es verdadero (`true`) entonces se ejecuta la *<secuencia_de_sentencias_v>*. Sin embargo, si su resultado es falso (`false`), entonces se ejecuta la *<secuencia_de_sentencias_f>*. Posteriormente la ejecución continúa por la siguiente intrucccción después de la sentencia `if`.

De esta forma, podríamos obtener el siguiente programa para convertir de pesetas a euros o viceversa, dependiendo del valor introducido por el usuario.

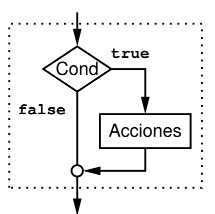
```
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;
int main()
{
   char resp ;
   cout << "Teclee P para convertir a Pesetas y E para convertir a Euros: " ;
   cin >> resp ;
   cout << "Introduce la cantidad : " ;
   double cantidad, result ;
   cin >> cantidad ;

   if (resp == 'P') {
      result = cantidad * EUR_PTS ;
   } else {
      result = cantidad / EUR_PTS ;
   }

   cout << cantidad << " equivale a " << result << endl ;
}
```

En determinadas ocasiones puede resultar interesante que el programa se plantee elegir entre ejecutar un grupo de sentencias o no hacer nada. En este caso la rama `else` puede ser omitida, obteniendo una sentencia de selección simple, que responde al siguiente esquema:

```
if ( <expresión_lógica> ) {
    <secuencia_de_sentencias> ;
}
```



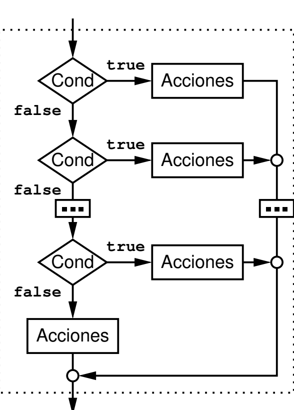
cuya ejecución comienza evaluando la expresión lógica. Si su resultado es verdadero (`true`), entonces se ejecuta la secuencia de sentencias entre llaves; si es falso (`false`), entonces no se ejecuta ninguna

sentencia. Como ejemplo, a continuación mostramos un programa que lee tres números por teclado e imprime el valor mayor:

```
#include <iostream>
using namespace std ;
int main ()
{
    int a, b, c ;
    cin >> a >> b >> c ;
    int mayor = a ;
    if (b > mayor) {
        mayor = b ;
    }
    if (c > mayor) {
        mayor = c ;
    }
    cout << mayor << endl ;
}
```

Es frecuente encontrar situaciones en las que se desea seleccionar una alternativa entre varias. En este caso se pueden encadenar varias sentencias de selección:

```
if ( <expresión_lógica_1> ) {
    <secuencia_de_sentencias_v1> ;
} else if ( <expresión_lógica_2> ){
    <secuencia_de_sentencias_v2> ;
} else if
    ...
    ...
} else {
    <secuencia_de_sentencias_f> ;
}
```



En este caso el comportamiento es el que se podría esperar intuitivamente. En primer lugar se evalúa la primera expresión lógica *<expresion_logica_1>*. Si el resultado es `true` se ejecuta la secuencia de sentencias *<secuencia_de_sentencias_v1>* y se descartan el resto de alternativas. Si el resultado es `false` se procede de igual manera con la siguiente expresión lógica *<expresion_logica_2>*, y así sucesivamente. La última rama, correspondiente a `else`, es opcional: si todas las expresiones lógicas fallan y aparece una rama `else` final, se ejecutan las sentencias *<secuencia_de_sentencias_f>*, pero que si no aparece, al no haber sentencias a ejecutar, el efecto es no ejecutar nada, continuando el flujo de control por la siguiente instrucción. Como ejemplo, el siguiente programa lee de teclado un número que representa una nota numérica entre cero y diez. Si es correcta queremos mostrar su equivalente en formato textual, y si es incorrecta queremos mostrar un mensaje de error.

```
#include <iostream>
using namespace std ;
int main ()
{
    double nota ;
    cin >> nota ;
    if ( ! ((nota >= 0.0) && (nota <= 10.0))) {
        cout << "Error: 0 <= n <= 10" << endl ;
    } else if (nota >= 9.5) {
        cout << "Matrícula de Honor" << endl ;
    } else if (nota >= 9.0) {
        cout << "Sobresaliente" << endl ;
```

```
    } else if (nota >= 7.0) {
        cout << "Notable" << endl ;
    } else if (nota >= 5.0) {
        cout << "Aprobado" << endl ;
    } else {
        cout << "Suspenso" << endl ;
    }
}
```

Sentencia switch

Aunque el uso de la sentencia `if` podría ser suficiente para seleccionar flujos de ejecución alternativos, hay numerosas situaciones que podrían quedar mejor descritas utilizando otras estructuras de control que se adapten mejor. Por ejemplo, supongamos que queremos ampliar nuestro programa de conversión de moneda para ofrecer un menú de opciones que permita convertir diferentes monedas (pesetas, francos, marcos y liras) a euros. En este caso se podría haber optado por usar una sentencia de selección múltiple, de la siguiente forma:

```
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;
const double EUR_FRC = 6.55957 ;
const double EUR_MRC = 1.95583 ;
const double EUR_LIR = 1936.27 ;
int main()
{
   char resp ;
   cout << "Teclee P para convertir de Pesetas a Euros" ;
   cout << "Teclee F para convertir de Francos a Euros" ;
   cout << "Teclee M para convertir de Marcos a Euros" ;
   cout << "Teclee L para convertir de Liras a Euros" ;
   cout << "Opcion: " ;
   cin >> resp ;
   cout << "Introduce la cantidad : " ;
   double cantidad, result ;
   cin >> cantidad ;

   if (resp == 'P'){
      result = cantidad * EUR_PTS ;
   }else if (resp == 'F'){
      result = cantidad * EUR_FRC ;
   }else if (resp == 'M'){
      result = cantidad * EUR_MRC ;
   }else { // Si no es ninguna de las anteriores es a Liras
      result = cantidad * EUR_LIR ;
   }

   cout << cantidad << " equivale a " << result << endl ;
}
```

Se trata de una situación en la que la secuencia de sentencias alternativas a ejecutar se decide en función del valor que tome una determinada expresión. En el ejemplo, la expresión es directamente una variable cuyo valor determina qué rama ejecutar. Para estas situaciones se dispone de la sentencia `switch`, cuyo uso hace que el programa quede más claro y, por tanto, mejor.

La sentencia `switch` tiene el siguiente formato y se utiliza en situaciones en que la expresión es de tipo *ordinal* (véase 2.2), y exige que los valores que se utilizan para seleccionar cada rama sean *constantes*.

```
switch ( <expresión> ) {
case <valor_cte_1> :
    <secuencia_de_sentencias_1> ;
    break ;
case <valor_cte_2> :
case <valor_cte_3> :
    <secuencia_de_sentencias_2> ;
    break ;
case <valor_cte_4> :
    <secuencia_de_sentencias_3> ;
    break ;
. . .
default:
    <secuencia_de_sentencias_d> ;
    break ;
}
```

Al ejecutar una sentencia `switch`, en primer lugar se evalúa la expresión de control. A continuación, se comprueba si el resultado coincide con alguno de los valores utilizados para seleccionar las diferentes ramas, comenzando por la primera. El flujo de ejecución continúa por aquella rama en la que se encuentre un valor que coincida con el valor resultado de evaluar la expresión de control. Si el valor de la expresión no coincide con ningún valor especificado, se ejecuta la secuencia de sentencias correspondiente a la etiqueta `default` (si es que existe). Aunque existen otras posibilidades, nosotros utilizaremos la sentencia `switch` en situaciones en las que queremos ejecutar una rama u otra de forma excluyente, por lo que después de la secuencia de sentencias asociada a cada rama **siempre** usaremos una sentencia `break`, cuya ejecución hace que el flujo de control pase a la sentencia del programa que se encuentra a continuación de la sentencia `switch`.

En nuestro ejemplo de conversión de múltiples monedas, podríamos haber escrito el programa usando `switch` de la siguiente forma:

```
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;
const double EUR_FRC = 6.55957 ;
const double EUR_MRC = 1.95583 ;
const double EUR_LIR = 1936.27 ;
int main()
{
   char resp ;
   cout << "Teclee P para convertir de Pesetas a Euros" ;
   cout << "Teclee F para convertir de Francos a Euros" ;
   cout << "Teclee M para convertir de Marcos a Euros" ;
   cout << "Teclee L para convertir de Liras a Euros" ;
   cout << "Opcion: " ;
   cin >> resp ;
   cout << "Introduce la cantidad : " ;
   double cantidad, result ;
   cin >> cantidad ;

   switch (resp){
   case 'P':
       result = cantidad * EUR_PTS ;
       break ;
   case 'F':
       result = cantidad * EUR_FRC ;
       break ;
   case 'M':
       result = cantidad * EUR_MRC ;
       break ;
   case 'L':
```

```
            result = cantidad * EUR_LIR ;
            break ;
        }

        cout << cantidad << " equivale a " << result << endl ;
    }
```

Aunque aparentemente la versión con `if` es similar a la versión con `switch`, en realidad esta segunda versión es más clara. Cualquier persona que lea el programa podrá suponer que su intención es seleccionar una alternativa u otra en función del valor resultado de evaluar una determinada expresión, mientras que para llegar a la misma conclusión en la versión con `if` es necesario leer una a una todas las expresiones que seleccionan la entrada en cada rama.

4.4. Sentencias de Iteración. Bucles

Las sentencias de iteración se utilizan para repetir la ejecución de un determinado grupo de sentencias dependiendo de determinadas condiciones. El flujo de ejecución estará iterando (por eso se les conoce como *bucles*), repitiendo la ejecución del grupo de sentencias (*cuerpo del bucle*) hasta que lo determine la expresión lógica utilizada para decidir la finalización del bucle.

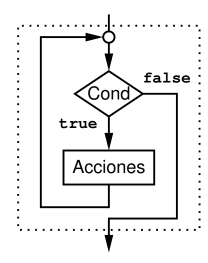
Un cálculo complejo suele construirse en base a repetir adecuadamente pasos más simples. Por ello la mayoría de los programas harán uso, de un modo u otro, de sentencias de iteración. Por este motivo es muy importante adquirir una destreza adecuada en el diseño de programas que hagan uso de estructuras de iteración.

Utilizaremos tres tipos de sentencias de iteración diferentes: `while`, `for` y `do-while`. En numerosas ocasiones será posible obtener la solución a un problema usando cualquiera de ellas. Sin embargo, deberemos procurar optar por aquel tipo de sentencia que mejor se adapte a la solución adoptada para el programa, ya que de esta forma conseguiremos programas más claros y mejores.

Sentencia while

Comenzamos mostrando la sentencia `while`, la más general, que responde al siguiente esquema:

```
while ( <expresión_lógica> ) {
    <secuencia_de_sentencias> ;
}
```



Su ejecución comienza con la evaluación de la expresión lógica. Si es falsa, el cuerpo del bucle no se ejecuta y el flujo de ejecución pasa directamente a la instrucción que siga a la sentencia *while*. Si es cierta, el flujo de ejecución pasa al cuerpo del bucle, el cual se ejecuta completamente en secuencia y posteriormente vuelve a evaluar la expresión lógica. Este ciclo iterativo consistente en evaluar la condición y ejecutar las sentencias se realizará *MIENTRAS* que la condición se evalúe a verdadero y finalizará cuando la condición se evalúe a falso.

Por ejemplo, supongamos que queremos que nuestro programa de conversión se ejecute repetidamente hasta que el usuario teclee 'N' como respuesta a la operación a realizar. En ese caso podríamos utilizar el siguiente programa:

```
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;
int main()
{
    char resp ;
```

```
        cout << "Teclee P (a Pesetas), E (a Euros) o F (Fin): " ;
        cin >> resp ;
        while (res != 'F'){
            cout << "Introduce la cantidad : " ;
            double cantidad, result ;
            cin >> cantidad ;
            if (resp == 'P'){
                result = cantidad * EUR_PTS ;
            }else{
                result = cantidad / EUR_PTS ;
            }
            cout << cantidad << " equivale a " << result << endl ;
            cout << "Teclee P (a Pesetas), E (a Euros) o F (Fin): " ;
            cin >> resp ;
        }
    }
```

A continuación, mostramos otro ejemplo en el que, dado un número tomado como entrada, escribimos en pantalla el primer divisor mayor que 1. Si el número no es mayor que 1 esto no es posible; en ese caso mostramos 1.

```
#include <iostream>
using namespace std ;
int main ()
{
    int num, divisor ;
    cin >> num ;
    if (num <= 1) {
        divisor = 1 ;
    } else {
        divisor = 2 ;
        while ((num % divisor) != 0) {
            ++divisor ;
        }
    }
    cout << "El primer divisor de " << num << " es " << divisor << endl ;
}
```

Como puede observarse en estos ejemplos, el número de iteraciones del bucle depende de cuando se evalúe a `false` la expresión lógica que, a su vez, depende de variables utilizadas en el bucle. Por ello, habrá que considerar qué variables utilizamos en la expresión de control y de qué forma se comportan. Para que el bucle se ejecute exactamente el número de iteraciones que nos interesa, es importante que nos aseguremos de que dichas variables tienen los valores adecuados inicialmente y que estudiemos cómo son modificadas en el cuerpo del bucle. Si las variables nunca fueran modificadas en el cuerpo tendríamos un *bucle infinito*, porque la condición de control nunca se cumpliría y por tanto el programa nunca terminaría.

Sentencia for

Frecuentemente nos encontramos con situaciones en las que el número de iteraciones que deseamos que ejecute el bucle es previsible y puede ser controlado utilizando una *variable de control*. En estos casos la solución queda más clara si utilizamos un tipo de bucle que permita conocer fácilmente qué variable se utiliza para controlar cuando acaba la ejecución, y cómo va evolucionando. Para ello, en C++ se utiliza la sentencia `for`, similar a la de otros lenguajes como Pascal o Modula-2, pero mucho más flexible, por lo que habrá que utilizarla aprovechando su flexibilidad pero sin abusar, porque ello podría producir programas inadecuados.

En realidad la sentencia `for` puede ser vista como una construcción especializada de la sentencia `while` presentada anteriormente, pero con una sintaxis diferente para hacer más explícito los casos

en los que la iteración está controlada por los valores que toma una determinada variable de control, de tal forma que existe una clara inicialización y una clara modificación (incremento) de la variable de control, hasta llegar al caso final. La sentencia `for` sigue el siguiente esquema:

```
for ( <inicialización> ; <expresión_lógica> ; <incremento> ) {
    <secuencia_de_sentencias> ;
}
```

y es equivalente a:

```
<inicialización> ;
while ( <expresión_lógica> ) {
    <secuencia_de_sentencias> ;
    <incremento> ;
}
```

Para utilizar adecuadamente la estructura `for` es necesario que el comportamiento iterativo del bucle quede claramente especificado utilizando únicamente los tres componentes (inicialización, condición de fin e incremento) de la cabecera de la sentencia. De esta forma, el programador podrá conocer el comportamiento del bucle sin necesidad de analizar el cuerpo del mismo. Así, nunca debemos modificar la variable de control de un bucle `for` dentro del cuerpo del mismo.

Nota: es posible y adecuado declarar e inicializar la variable de control del bucle en el lugar de la inicialización. En este caso especial, el ámbito de visibilidad de la variable de control del bucle es solamente hasta el final del bloque de la estructura `for`.

Como ejemplo, a continuación mostramos un programa que toma como entrada un número `n` y muestra por pantalla la serie de números 0 1 2 . . . n.

```
#include <iostream>
using namespace std ;
int main ()
{
    int n ;
    cin >> n ;
    for (int i = 0 ; i < n ; ++i) {
        cout << i << " " ;
    }
    // i NO es visible aquí
    cout << endl ;
}
```

Ya hemos visto cómo la sentencia `for` puede ser considerada como una especialización de la sentencia `while`, por lo que el programa anterior también podría haber sido descrito en base a la sentencia `while` de la siguiente forma:

```
#include <iostream>
using namespace std ;
int main ()
{
    int n ;
    cin >> n ;

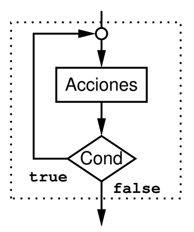
    int i = 0;
    while (i < n) {
        cout << i << " " ;
        ++i;
    }
    // i SI es visible aquí
    cout << endl ;
}
```

Ambas soluciones son correctas y a primera vista puede parecer que de similar claridad. Sin embargo, la primera solución es mejor que la segunda: en el primer caso toda la información necesaria para analizar el comportamiento del bucle está claramente localizable, mientras que en el segundo es necesario localizarla estudiando cómo se inicializa la variable y cómo se incrementa. Aunque en este ejemplo ello está claro, piense que en otras situaciones esta información puede estar mucha más oculta y, por tanto, ser más difícil de obtener.

Sentencia do-while

Finalmente, se dispone de un tercer tipo de sentencia de iteración, la sentencia `do-while`. Al igual que ocurre con la sentencia `for`, esta sentencia puede ser vista como una construcción especializada para determinadas situaciones que, aunque admiten una solución en base a la sentencia `while`, presentan una solución más clara con `do-while`. La sentencia `do-while` sigue el siguiente esquema:

```
do {
    <secuencia_de_sentencias> ;
} while ( <expresión_lógica> ) ;
```



En este caso, a diferencia de la sentencia `while`, en la que primero se evalúa la expresión lógica y después, en caso de ser cierta, se ejecuta la secuencia de sentencias, en la estructura `do-while` primero se ejecuta la secuencia de sentencias y posteriormente se evalúa la expresión lógica y, si ésta es cierta, se repite el proceso. En este caso, el flujo de ejecución alcanza la expresión lógica después de ejecutar el cuerpo del bucle *al menos una vez*. Ello hace que se utilice para situaciones en las que sabemos que siempre queremos que el cuerpo del bucle se ejecute al menos una vez. Por ejemplo, si queremos diseñar un programa que lea un número y se asegure de que es par, repitiendo la lectura en caso de no serlo, podríamos hacer lo siguiente:

```
#include <iostream>
using namespace std;
int main ()
{
    int num ;
    do {
        cout << "Tecle un número par: " ;
        cin >> num ;
    } while ((num % 2) != 0) ;
    cout << "El número par es " << num << endl ;
}
```

Como hemos mencionado, también sería posible obtener una versión alternativa de este programa haciendo uso de una sentencia `while`, de la siguiente forma:

```
#include <iostream>
using namespace std;
int main ()
{
    int num ;
    cout << "Tecle un número par: " ;
    cin >> num ;

    while ((num % 2) != 0){
        cout << "Tecle un número par: " ;
        cin >> num ;
    }
```

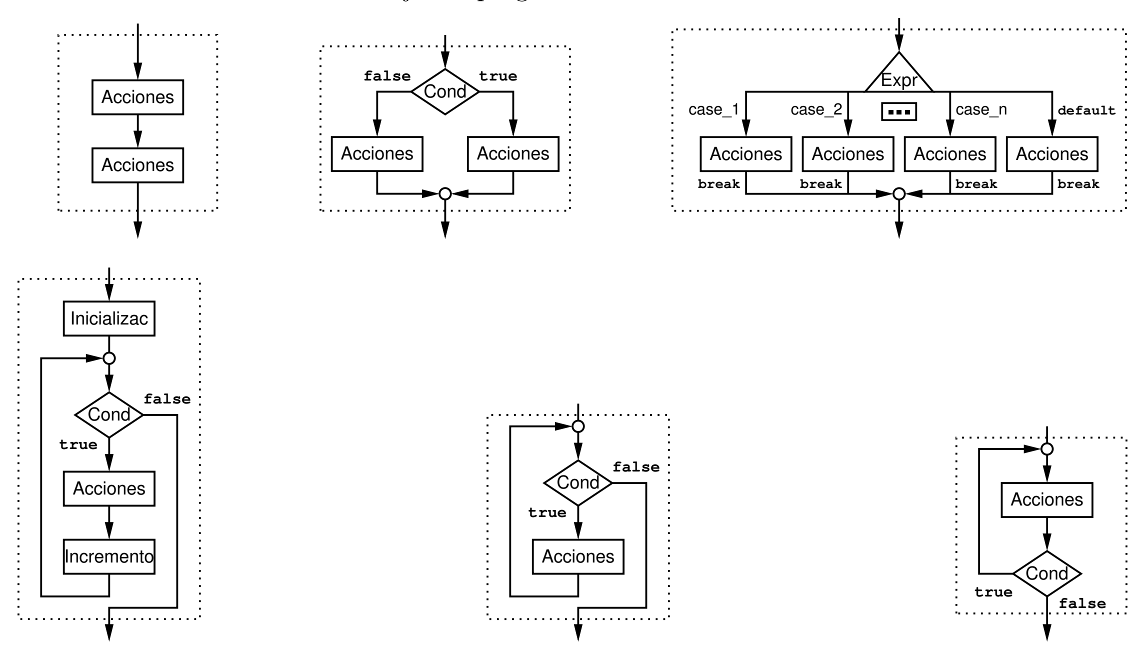
```
        cout << "El número par es " << num << endl ;
    }
```

Sin embargo, como puede observarse, la primera versión se adapta mejor al problema a resolver. Si ya sabemos que el cuerpo del bucle se va a repetir al menos una vez, ¿por qué no utilizar un tipo de sentencia de iteración que ya lo expresa? El segundo programa se basa en *sacar una iteración* fuera del bucle y utilizar un tipo de bucle que se repite cero o más veces. Aunque el comportamiento es equivalente, la solución parece un poco forzada.

4.5. Programación Estructurada

Un programa sigue una metodología de programación estructurada si todas las estructuras de control que se utilizan (secuencia, selección, iteración y modularización) tienen un único punto de entrada y un único punto de salida. Esta característica hace posible que se pueda aplicar la *abstracción* para su diseño y desarrollo. La abstracción se basa en la identificación de los elementos a un determinado nivel, ignorando los detalles especificados en niveles inferiores. Un algoritmo que use tan solo las estructuras de control tratadas en este tema, se dice que está *estructurado*.

Bohm y Jacopini demostraron que todo problema computable puede resolverse usando únicamente estas estructuras de control. Ésta es la base de la programación estructurada. El diseño de programas de acuerdo al principio de programación estructurada permite aplicar el principio de abstracción, tanto a la hora de diseñar el programa como a la hora de leer programas de otros, lo cual facilita enormemente el trabajo del programador.



Para que la aplicación del principio sea factible, es necesario que el programador pueda confiar en que cada estructura tiene un único punto de entrada y de salida. Ello es especialmente importante cuando se trabaja con subprogramas (ver el capítulo 5), pero también es importante al trabajar con las estructuras básicas presentadas en este capítulo. Para aclarar este punto volveremos a repasar la sentencia de iteración `for`. Sabemos que su gran ventaja está en que permite escribir en una misma zona del programa toda la información necesaria para saber cómo evoluciona el bucle. Ello es así porque dentro de la misma sentencia se encuentra la variable de control utilizada en el mismo, su inicialización y su modificación, después de cada iteración. El programador puede leer esta zona del programa y conocer cuántas iteraciones va a dar el bucle, utilizando la abstracción, ignorando el cuerpo del bucle. En realidad, ello es así si estamos seguros de que el programador no hace ninguna modificación de la variable de control dentro del cuerpo del bucle. Si el programador admite la posibilidad de que la variable de control sea modificada dentro del cuerpo del bucle,

entonces es imposible aplicar el principio de abstracción y, por tanto, el programador se ve forzado a leer con detalle el cuerpo del bucle para estar seguro del número de iteraciones que produce.

Por este motivo, seguiremos como principio básico el siguiente criterio de estilo en el diseño de nuestros programas: **nunca modificaremos la variable de control dentro de un bucle `for`**. Así mismo, tampoco se podrá modificar ninguna de las variables que aparecen en la condición de finalización.

4.6. Ejemplos

A continuación, repasaremos los diferentes elementos introducidos en este capítulo viendo algunos ejemplos.

Ejemplo 1. Producto de dos números

Queremos realizar un programa que lea por teclado dos números y muestre por pantalla su producto. Como queremos que este ejemplo nos sirva como ejemplo de estructuras de iteración, supondremos que no es posible utilizar el operador de multiplicación (`*`) directamente. En primer lugar necesitamos tener una solución al problema y en segundo lugar necesitamos escribir adecuadamente nuestra solución en C++. Aunque todos conocemos las tablas de multiplicar, también sabemos que una multiplicación no es más que una serie de sumas sucesivas de uno de los operandos tantas veces como indique el otro (`n x m = m+m+m+...+m` n veces). Multiplicar 3 x 2 no es más que sumar 3 veces el número 2. En este momento ya tenemos la solución y tenemos que escribirla en C++. Vemos que hay un proceso que se repite y que este proceso siempre se repite n veces. En tal caso, resulta adecuado usar un bucle `for`, controlado por una variable que asegure que se ejecutan exactamente `n` iteraciones. En cada iteración bastará con acumular una nueva suma del valor `m` al total. Como resultado de todo ello, obtenemos el siguiente programa:

```
#include <iostream>
using namespace std ;

int main ()
{
    cout << "Introduzca dos números: " ;
    int m, n ;
    cin >> m >> n ;

    int total = 0 ;
    for (int i = 0 ; i < n ; ++i) {
        // Proceso iterativo: acumular el valor de 'm' al total
        total = total + m ;     // total += m ;
    }
    cout << total << endl ;
}
```

Ejemplo 2. Factorial

A continuación queremos obtener un programa para calcular el factorial de un número dado como entrada (`n! = 1 * 2 * ... * n`). De nuevo se trata de un problema similar al anterior: hay un proceso iterativo, en el que conocemos el número de iteraciones a realizar y un procesamiento acumulativo, en el que en cada iteración multiplicamos por un valor diferente. En este caso, como también conocemos las iteraciones que queremos que ejecute el bucle, volvemos a usar una sentencia `for`, pero en esta ocasión hacemos que la variable de control `i` evolucione conteniendo valores que puedan ser aprovechados dentro del cuerpo del bucle, haciendo que su rango de valores vaya desde 2 hasta `n`. Nótese que en esta ocasión utilizamos el valor de la variable de control dentro del cálculo iterativo.

```
#include <iostream>
using namespace std ;

int main ()
{
    cout << "Introduzca un número: " ;
    int n ;
    cin >> n ;
    // Multiplicar: 1 2 3 4 5 6 7 ... n
    int fact = 1 ;
    for (int i = 2 ; i <= n ; ++i) {
        // Proceso iterativo: acumular el valor de 'i' al total
        fact = fact * i ;       // fact *= i ;
    }
    cout << fact << endl ;
}
```

Ejemplo 3. División entera

Finalmente, mostraremos un programa que, dados dos números tomados como entrada, muestra por pantalla su división entera (cociente y resto). De nuevo, en este programa evitaremos utilizar los operadores predefinidos disponibles en C++ para ello (`/` y `%`). En su lugar, aprovecharemos que la división entera no es más que el resultado de una serie de restas sucesivas del divisor al dividendo, hasta que no sea posible continuar. Nuestro programa comprueba que el divisor no es cero, mostrando un mensaje de error en caso contrario. Usamos una sentencia `while` porque, a diferencia con lo que ocurría en el ejercicio del producto de dos números, ahora no sabemos cuántas iteraciones se necesitarán para resolver el problema.

```
#include <iostream>
using namespace std ;
int main ()
{
    cout << "Introduzca dos números: " ;
    int dividendo, divisor ;
    cin >> dividendo >> divisor ;
    if (divisor == 0) {
        cout << "El divisor no puede ser cero" << endl ;
    } else {
        int resto = dividendo ;
        int cociente = 0 ;
        while (resto >= divisor) {
            resto -= divisor ;
            ++cociente ;
        }
        cout << cociente << " " << resto << endl ;
    }
}
```

Pruebe a modificar el programa anterior para que siempre sea posible realizar la división. Para ello, modifique el proceso de entrada de forma que se asegure de que el dividendo es mayor o igual que cero y el divisor es mayor que cero. Si alguna de las dos cosas no ocurre, repita la toma de datos hasta que ambos datos sean correctos.

Capítulo 5

Subprogramas. Funciones y Procedimientos

El diseño de un programa es una tarea compleja, por lo que deberemos abordarla siguiendo un enfoque que permita hacerla lo más simple posible. Al igual que ocurre en otros contextos, la herramienta fundamental para abordar la solución de problemas complejos es la *abstracción*: una herramienta mental que nos permite tratar el problema identificando sus elementos fundamentales y dejando para más adelante el estudio de los detalles secundarios. La aplicación de este principio al desarrollo de programas permite seguir un enfoque de refinamientos sucesivos: en cada fase del diseño del programa ignoramos los detalles secundarios y nos centramos en lo que nos interesa en ese momento; en fases posteriores abordamos los detalles que hemos ignorado por el momento. De esta forma, al final tenemos un diseño completo, obtenido con menor esfuerzo y de forma más segura.

Los lenguajes de programación ofrecen la posibilidad de definir subprogramas, permitiendo al programador aplicar explícitamente la abstracción en el diseño y construcción de software. Un subprograma puede ser visto como un *mini* programa encargado de resolver un subproblema, que se encuentra englobado dentro de otro mayor. En ocasiones también pueden ser vistos como una ampliación del conjunto de operaciones básicas del lenguaje de programación, proporcionándole nuevas herramientas no disponibles de forma *predefinida*.

5.1. Funciones y Procedimientos

Un subprograma codifica la solución algorítmica a un determinado problema. Cuando en un determinado momento de la computación es necesario resolver dicho problema se hace uso de dicho subprograma mediante una invocación (*llamada*) al mismo. Por lo general, la resolución del problema requerirá proporcionar al subprograma la información necesaria para abordar su solución (de entrada al subprograma), y es posible que el subprograma responda con algún resultado (de salida del subprograma). Para ello, un subprograma se comunica con el que lo invoca (llama) mediante *parámetros*.

Podemos distinguir dos tipos de subprogramas:

- ***Procedimientos***: encargados de resolver un problema computacional general. En el siguiente ejemplo se dispone de dos variables `x` e `y`, con determinados valores y hacemos una invocación al procedimiento `ordenar` para conseguir que, como resultado, el menor de dichos números quede almacenado en la variable `x` y el mayor en la variable `y`. Como vemos, en este momento no nos preocupa qué hay que hacer para que la ordenación realmente tenga efecto, estamos ignorando esos detalles. Suponemos que el subprograma `ordenar` se encargará de ello. El lenguaje C++ no dispone del operador `ordenar`, por lo que en algún momento del diseño deberemos refinar esta solución y proporcionar una definición para dicho subprograma.

```
int main()
{
    int x = 8 ;
    int y = 4 ;
    ordenar(x, y) ;
    cout << x << " " << y << endl ;
}
```

Como vemos, la función `main` se comunica con el subprograma `ordenar` mediante una *llamada*. En esta comunicación hay un intercambio de información entre `main` y `ordenar` a través de los parámetros utilizados en la llamada, en este caso las variables `x` e `y`.

- ***Funciones***: encargadas de realizar un cálculo computacional y generar un único resultado. Las funciones responden a los mismos principios que los procedimientos, salvo que están especializados para que la comunicación entre el que hace la invocación y la función llamada tenga lugar de una forma especial, que se adapta muy bien y es muy útil en numerosas situaciones. En el siguiente ejemplo, la función `calcular_menor` recibe dos números como parámetros y calcula el menor de ellos. En este caso la comunicación entre el que hace la llamada (la función `main`) y la función llamada se hace de forma diferente. Antes hicimos la invocación en una instrucción independiente, sin embargo, ahora se hace como parte de una instrucción más compleja. Ello es así porque una función *devuelve* un valor (en este caso el menor número) y dicho valor deberá ser utilizado como parte de un cálculo más complejo. En nuestro ejemplo, como resultado de la invocación obtendremos un número que será almacenado en una variable de la función `main`.

```
int main()
{
    int x = 8 ;
    int y = 4 ;
    int z = calcular_menor(x, y) ;
    cout << "Menor: " << z << endl ;
}
```

En los ejemplos planteados vemos que tanto procedimientos como funciones se utilizan para realizar un cálculo, ignorando los detalles y, por tanto, simplificando el diseño del programa principal. En ambos casos hay una invocación al subprograma correspondiente y un intercambio de información entre el que llama y el subprograma llamado. La única diferencia entre ambos tipos de subprogramas está en la forma de hacer las llamadas:

- La llamada a un procedimiento constituye por sí sola una sentencia independiente que puede ser utilizada como tal en el cuerpo de otros subprogramas (y del programa principal). La única forma de intercambiar información es a través de los parámetros usados en la llamada.
- La llamada a una función **no** constituye por sí sola una sentencia, por lo que debe aparecer dentro de alguna sentencia que utilice el valor resultado de la función. La información se intercambia a través de los parámetros y mediante el valor devuelto por la función.

5.2. Definición de Subprogramas

Si intentáramos compilar los programas utilizados anteriormente como ejemplo, comprobaríamos que el compilador nos informa de un mensaje de error. Ello es así porque no sabe a qué nos referimos al hacer uso de `ordenar` y de `calcular_menor`. Se trata de dos identificadores nuevos, que no se encuentran predefinidos en C++ y que, por lo tanto, deben ser declarados antes de su uso, como ocurre con cualquier identificador nuevo. Nótese que ambos nombres de subprogramas corresponden a tareas que el programador supone, pero cuya solución no ha sido descrita aún en ninguna parte del programa. Por este motivo, es necesario proporcionar una definición precisa de los subprogramas `ordenar` y `calcular_menor` que, además, debe estar situada en un punto del

programa que la haga *visible* en el punto en que se usen. Por ahora situaremos la definición de los subprogramas antes de su uso, aunque en la sección 5.7 mostraremos otras posibilidades.

Hemos comentado que un subprograma es como un *mini* programa encargado de resolver un subproblema, por lo que la definición de un subprograma no difiere de la definición ya utilizada de la función `main`, que en realidad no es más que una función especial. A continuación se muestra un programa completo con la definición de la función `main`, que define el comportamiento del programa principal, y las definiciones de los dos subprogramas `ordenar` y `calcular_menor`, utilizados desde la función `main`.

```
#include <iostream>
using namespace std ;
int calcular_menor(int a, int b)
{
    int menor ;
    if (a < b) {
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
void ordenar(int& a, int& b)
{
    if (a > b) {
        int aux = a ;
        a = b ;
        b = aux ;
    }
}
int main()
{
    int x = 8 ;
    int y = 4 ;
    int z = calcular_menor(x, y) ;
    cout << "Menor: " << z << endl ;
    ordenar(x, y) ;
    cout << x << " " << y << endl ;
}
```

La definición de un subprograma consta de un encabezamiento y de un cuerpo. En el encabezamiento se especifica su nombre, su tipo y los parámetros con los que dicho subprograma se comunica con el exterior. En el cuerpo se describe la secuencia de acciones necesarias para conseguir realizar la tarea que tiene asignada.

El encabezamiento comienza con el nombre del tipo devuelto por el subprograma. Dicho tipo será `void` si se trata de un procedimiento, ya que en dicho tipo de subprogramas no se devuelve ningún valor. Si se trata de una función, dicho tipo será el correspondiente al valor devuelto por la misma.

La definición de un subprograma se hace de forma independiente de las llamadas en las que se haga uso del mismo. Por este motivo, debe contener la declaración de todos los elementos que use. En el caso de los parámetros, habrá que especificar el nombre con el que nos referimos a ellos en el cuerpo del subprograma y su tipo.

El cuerpo del subprograma especifica la secuencia de acciones a ejecutar necesarias para resolver el subproblema especificado, y podrá definir tantas variables *locales* como necesite para desempeñar su misión. En el caso de una función, el valor que devuelve (el valor que toma tras la llamada) vendrá dado por el resultado de evaluar la expresión de la sentencia `return`. Aunque C++ es más flexible, nosotros sólo permitiremos una única utilización de la sentencia `return` y deberá ser al final del cuerpo de la función. Así mismo, un procedimiento no tendrá ninguna sentencia `return` en su cuerpo.

5.3. Ejecución de Subprogramas

Cuando se produce una llamada (invocación) a un subprograma:

1. Se establecen las vías de comunicación entre los algoritmos llamante y llamado por medio de los parámetros.

2. Posteriormente, el flujo de ejecución pasa a la primera sentencia del cuerpo del subprograma llamado, cuyas instrucciones son ejecutadas secuencialmente, en el orden en que están escritas.

3. Cuando sea necesario, se crean las variables locales especificadas en el cuerpo del subprograma.

4. Cuando finaliza la ejecución del subprograma, las variables locales y los parámetros previamente creados se destruyen, el flujo de ejecución retorna al (sub)programa llamante, y continúa la ejecución por la sentencia siguiente a la llamada realizada.

5.4. Paso de Parámetros. Parámetros por Valor y por Referencia

El intercambio de información en la llamada a subprogramas no difiere del intercambio de información en una comunicación general entre dos entidades. La información puede fluir en uno u otro sentido, o en ambos. Dicho intercambio de información se realiza a través de los parámetros. Dependiendo del contexto en que se utilicen, hablaremos de *parámetros formales*, si nos referimos a los que aparecen en la definición del subprograma, o de *parámetros actuales (o reales)*, si nos referimos a los que aparecen en la llamada al subprograma. A continuación analizaremos este intercambio de información, *visto desde el punto de vista del subprograma llamado*, y mostraremos cómo habrá que definir los parámetros formales en cada caso.

- Serán *parámetros de entrada* aquellos que se utilizan para recibir la información necesaria para realizar una computación. Por ejemplo, los parámetros `a` y `b` de la función `calcular_menor` anterior.

  Los parámetros de entrada se definen mediante *paso por valor* (cuando son de tipos simples[1]). Ello significa que los parámetros formales son variables independientes, que toman sus valores iniciales como *copias* de los valores de los parámetros actuales de la llamada en el momento de la invocación al subprograma. El parámetro actual puede ser cualquier expresión cuyo tipo sea compatible con el tipo del parámetro formal correspondiente. Se declaran especificando el tipo y el identificador asociado.

```
int calcular_menor(int a, int b)
{
    int menor ;
    if (a < b) {
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
```

- Denominamos *parámetros de salida* a aquellos que se utilizan para transferir al programa llamante información producida como parte de la computación/solución realizada por el subprograma.

[1]Cuando son de tipos compuestos (véase 6) se definen mediante *paso por referencia constante*, que será explicado más adelante.

Los parámetros de salida se definen mediante *paso por referencia*. Ello significa que el parámetro formal es una *referencia a la variable* que se haya especificado como parámetro actual en el momento de la llamada al subprograma. Ello exige que el parámetro actual correspondiente a un parámetro formal por referencia sea una **variable**. Cualquier acción dentro del subprograma que se haga sobre el parámetro formal se realiza sobre la variable referenciada, que aparece como parámetro actual en la llamada al subprograma. Se declaran especificando el tipo, el símbolo "*ampersand*" (`&`) y el identificador asociado.

En el siguiente ejemplo, el procedimiento `dividir` recibe, en los parámetros de entrada `dividendo` y `divisor`, dos números que queremos dividir para obtener su cociente y resto. Dichos resultados serán devueltos al punto de la llamada utilizando los parámetros de salida `coc` y `resto`, que son pasados por referencia.

```
void dividir(int dividendo, int divisor, int& coc, int& resto)
{
    coc = dividendo / divisor ;
    resto = dividendo % divisor ;
}
int main()
{
    int cociente ;
    int resto ;

    dividir(7, 3, cociente, resto) ;
    // ahora 'cociente' valdrá 2 y 'resto' valdrá 1
}
```

En realidad, dado que los parámetros `coc` y `resto` han sido pasados por referencia, una vez efectuada la llamada éstos quedan asociados a las correspondientes variables usadas como parámetros actuales (las variables `cociente`, `resto` de la función `main`). Así, cualquier modificación en los parámetros es, en realidad, una modificación de las variables `cociente` y `resto` de la función `main`. De esta forma, cuando acaba el subprograma conseguimos que los resultados estén en las variables adecuadas del llamante. El efecto es como si se hubiera producido una *salida* de resultados desde el subprograma llamado hacia el subprograma que realizó la llamada.

- Denominamos *parámetros de entrada/salida* a aquellos que se utilizan tanto para recibir información de entrada, necesaria para que el subprograma pueda realizar su computación, como para devolver los resultados obtenidos de la misma. Se definen mediante *paso por referencia* y se declaran como se especificó anteriormente para los parámetros de salida.

  Por ejemplo, los parámetros `a` y `b` del procedimiento `ordenar` son utilizados tanto de entrada como de salida. En el momento de la llamada aportan como entrada las variables que contienen los valores a ordenar y, cuando acaba el subprograma, son utilizados para devolver los resultados. Ello es así porque se definen mediante paso por referencia. De esta forma, cuando dentro del subprograma trabajamos con los parámetros formales `a` y `b`, en realidad accedemos a las variables utilizadas en la llamada. Si dentro del subprograma se intercambian los valores de los parámetros, indirectamente se intercambian también los valores de las variables de la llamada. Al terminar el subprograma el resultado está en las variables utilizadas en la llamada.

```
void ordenar(int& a, int& b)
{
    if (a > b) {
        int aux = a ;
        a = b ;
        b = aux ;
    }
}
```

La siguiente tabla relaciona los diferentes modos de comunicación con la forma de efectuar el paso de parámetros. Todo lo comentado hasta ahora es aplicable a tipos simples. En el capítulo 6 consideraremos el paso de parámetros para el caso de tipos compuestos.

	Tipos	
	Simples	Compuestos
(⇓) Entrada	P.Valor `(int x)`	P.Ref.Cte `(const Persona& x)`
(⇑) Salida, (⇕) E/S	P.Ref `(int& x)`	P.Ref `(Persona& x)`

Reglas a seguir en el paso de parámetros

En la llamada a un subprograma se deben cumplir las siguientes normas:

- El número de parámetros actuales debe coincidir con el número de parámetros formales.
- Cada parámetro formal se corresponde con aquel parámetro actual que ocupe la misma posición en la llamada.
- El tipo del parámetro actual debe estar acorde con el tipo del correspondiente parámetro formal.
- Un parámetro formal de salida o entrada/salida (paso por referencia) requiere que el parámetro actual sea una variable.
- Un parámetro formal de entrada (paso por valor o paso por referencia constante) permite que el parámetro actual sea una variable, constante o cualquier expresión.

5.5. Criterios de Modularización

No existen métodos objetivos para determinar como descomponer la solución de un problema en subprogramas, es una labor subjetiva. No obstante, se siguen algunos criterios que pueden guiarnos para descomponer un problema y modularizar adecuadamente. El diseñador de software debe buscar un **bajo acoplamiento** entre los subprogramas y una **alta cohesión** dentro de cada uno.

- Acoplamiento: Un objetivo en el diseño descendente es crear subprogramas aislados e independientes. Sin embargo, debe haber alguna conexión entre los subprogramas para formar un sistema coherente. A dicha conexión se conoce como acoplamiento. Por lo tanto, maximizar la independencia entre subprogramas será minimizar el acoplamiento.
- Cohesión: Hace referencia al grado de relación entre las diferentes partes internas dentro de un mismo subprograma. Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro del subprograma es tal que posteriores modificaciones podrán resultar complicadas. Se busca maximizar la cohesión dentro de cada subprograma

Si no es posible analizar y comprender un subprograma de forma aislada e independiente del resto, entonces podemos deducir que la división modular no es la más adecuada.

5.6. Subprogramas "en Línea"

La llamada a un subprograma conlleva un pequeño coste debido al control y gestión de la misma que ocasiona cierta pérdida de tiempo de ejecución.

Hay situaciones en las que el subprograma es tan pequeño que el coste asociado a la invocación es superior al coste asociado a computar la solución del mismo, de tal forma que en estas situaciones interesa eliminar el coste asociado a su invocación. En ese caso se puede especificar que el

subprograma se traduzca como *código en línea* en vez de como una llamada a un subprograma. Para ello se especificará la palabra reservada `inline` justo antes del tipo. De esta forma, se mantiene los beneficios proporcionados por la abstracción, pero se eliminan los costes asociados a la invocación.

```
inline int calcular_menor(int a, int b)
{
    return (a < b) ? a : b ;
}
```

Este mecanismo sólo es adecuado cuando el cuerpo del subprograma es muy pequeño, de tal forma que el coste asociado a la invocación dominaría respecto a la ejecución del cuerpo del mismo.

5.7. Declaración de Subprogramas. Prototipos

Hemos comentado que, al igual que cualquier nuevo elemento del programa al que hagamos referencia con un identificador (constante simbólica, variable o tipo), los subprogramas también deben ser declarados, permitiendo así que no resulten desconocidos para el compilador.

La definición de un subprograma sirve también como declaración del mismo. Hasta ahora hemos seguido el criterio de definir el subprograma antes de su uso, pero también existe la posibilidad de efectuar una declaración del subprograma, sin proporcionar su definición completa. En este caso basta con declarar el *prototipo* del subprograma y suponer que éste será definido posteriormente. El prototipo de un subprograma está formado por su encabezamiento (sin incluir el cuerpo del mismo), y acabado en el delimitador "punto y coma" (`;`). Una vez declarado el prototipo de un subprograma su ámbito de visibilidad será global al fichero, es decir, desde el lugar en que ha sido declarado hasta el final del fichero en el que aparece.

```
int calcular_menor(int a, int b) ;  // prototipo de 'calcular_menor'
int main()
{
    int x = 8 ;
    int y = 4 ;
    int z = calcular_menor(x, y) ;
}
int calcular_menor(int a, int b)   // definición de 'calcular_menor'
{
    int menor ;
    if (a < b) {
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
```

5.8. Sobrecarga de Subprogramas

Se denomina sobrecarga al uso del mismo identificador para aludir a subprogramas u operadores diferentes. Para saber a cuál de los diferentes subprogramas u operadores nos estamos refiriendo se utilizan los parámetros de los mismos, que deberán ser diferentes, ya que en otro caso el compilador no sería capaz de determinar a cual nos referimos con una determinada llamada. A continuación mostramos ejemplos de subprogramas `imprimir` que están sobrecargados y pueden ser utilizados para imprimir números de tipo `int` y `double`.

```
void imprimir(int x)
{
```

```
    cout << "entero: " << x << endl ;
}
void imprimir(double x)
{
    cout << "real: " << x << endl ;
}
```

En el siguiente ejemplo la función `media` permite calcular la media de dos o de tres números enteros, dependiendo de cómo se efectúe la llamada.

```
double media(int x, int y, int z) {
    return double(x + y + z) / 3.0 ;
}
double media(int x, int y)
{
    return double(x + y) / 2.0 ;
}
```

5.9. Pre-Condiciones y Post-Condiciones

- Pre-condición es un enunciado que debe ser cierto antes de la llamada a un subprograma. Especifica las condiciones bajo las cuales se ejecutará dicho subprograma.
- Post-condición es un enunciado que debe ser cierto tras la ejecución de un subprograma. Especifica el comportamiento de dicho subprograma.
- Codificar las pre/post-condiciones mediante asertos proporciona una valiosa documentación, y tiene varias ventajas:
  - Hace al programador explícitamente consciente de los prerrequisitos y del objetivo del subprograma.
  - Durante la depuración, las pre-condiciones comprueban que la llamada al subprograma se realiza bajo condiciones validas.
  - Durante la depuración, las post-condiciones comprueban que el comportamiento del subprograma es adecuado.
  - Sin embargo, a veces no es posible codificarlas fácilmente.

En C++, las pre-condiciones y post-condiciones se pueden especificar mediante asertos, para los cuales es necesario incluir la biblioteca `cassert`. Por ejemplo:

```
#include <iostream>
#include <cassert>
using namespace std ;
//---------------------------
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    assert(divisor != 0) ;                              // PRE-CONDICION
    cociente = dividendo / divisor ;
    resto = dividendo % divisor ;
    assert(dividendo == (divisor * cociente + resto)) ; // POST-CONDICION
}
```

Nota: en *GNU GCC* es posible desactivar la comprobación de asertos mediante la siguiente directiva de compilación:

```
g++ -DNDEBUG -ansi -Wall -Werror -o programa programa.cpp
```

5.10. Ejemplo. Números primos

Finalizaremos el capítulo mostrando el diseño de un programa completo, en el que pretendemos imprimir todos los números primos de un cierto intervalo, dado por dos valores leídos por teclado. En primer lugar leemos los números y nos aseguramos de que definan un intervalo correcto. Para ello, hacemos uso del subprograma `ordenar`, que sitúa los números de forma que el valor contenido en la primera variable sea menor que el de la segunda. Una vez que disponemos de un intervalo válido, hacemos uso de un subprograma `primos`, que muestra por pantalla todos los primos de un cierto intervalo, definido por dos parámetros de entrada. El subprograma `primos` prueba uno a uno con todos los números del intervalo comprobando si se trata de un número primo.

La tarea de ver si un número es primo representa una abstracción operacional claramente definida, por lo que conviene definir un nuevo subprograma que se encargue de dicha tarea de forma independiente de la implementación del subprograma `primos`. Para ello, definimos una función `es_primo` que, dado un número como entrada devuelve un valor (de tipo `bool`) indicando si es primo (`true`) o no (`false`).

```
//- fichero: primos.cpp --------------------------------------------
#include <iostream>
using namespace std ;

void ordenar(int& menor, int& mayor)
{
   if (mayor < menor) {
       int aux = menor ;
       menor = mayor ;
       mayor = aux ;
   }
}
bool es_primo(int x)
{
   unsigned i = 2;

   while ((i <= x/2) && ( x % i != 0)) {
       i++;
   }
   return (i == x/2+1) ;
}
void primos(int min, int max)
{
   cout << "Números primos entre " << min << " y " << max << endl ;
   for (int i = min ; i <= max ; ++i) {
       if (es_primo(i)) {
           cout << i << " " ;
       }
   }
   cout << endl ;
}
int main()
{
   int min, max ;
   cout << "Introduzca el rango de valores " ;
   cin >> min >> max ;
   ordenar(min, max) ;
   primos(min, max) ;
}
//- fin: primos.cpp ------------------------------------------------
```

Capítulo 6

Tipos Compuestos

Los *tipos compuestos* surgen de la composición y/o agregación de otros tipos para formar nuevos tipos de mayor entidad. Existen dos formas fundamentales de crear tipos de mayor entidad: la composición de elementos, que denominaremos "Registros" o "Estructuras" y la agregación de elementos del mismo tipo, que se conocen como "Agregados", "Arreglos" o mediante su nombre en inglés "Arrays". Además de estos tipos compuestos definidos por el programador, los lenguajes de programación suelen proporcionar algún tipo adicional para representar "cadenas de caracteres".

6.1. Paso de Parámetros de Tipos Compuestos

Los lenguajes de programación normalmente utilizan el *paso por valor* y el *paso por referencia* para implementar la transferencia de información entre subprogramas descrita en el interfaz. Para la transferencia de información de *entrada*, el paso por valor supone *duplicar y copiar el valor* del parámetro actual en el formal. En el caso de tipos *simples*, el paso por valor es adecuado para la transferencia de información de *entrada*, sin embargo, si el tipo de dicho parámetro es *compuesto*, es posible que dicha copia implique una *alta sobrecarga*, tanto en espacio de memoria como en tiempo de ejecución. El lenguaje de programación *C++* permite realizar de forma eficiente la transferencia de información de **entrada** para tipos compuestos mediante el *paso por referencia constante*.

Así, en el *paso por referencia constante* el parámetro formal es una referencia al parámetro actual especificado en la llamada, tomando así su valor, pero no puede ser modificado al ser una referencia constante, evitando de esta forma la semántica de salida asociada al paso por referencia. El paso por referencia constante suele utilizarse para el paso de parámetros de entrada con tipos compuestos, ya que evita la duplicación de memoria y la copia del valor, que en el caso de tipos compuestos suele ser costosa. Para ello, los parámetros se declaran como se especificó anteriormente para el paso por referencia, pero anteponiendo la palabra reservada `const`.

```
void imprimir(const Fecha& fech)
{
    cout << fech.dia << (int(fech.mes)+1) << fech.anyo << endl ;
}
```

	Tipos	
	Simples	Compuestos
($\Downarrow$) Entrada	P.Valor `(int x)`	P.Ref.Cte `(const Persona& p)`
($\Uparrow$) Salida, ($\Updownarrow$) E/S	P.Ref `(int& x)`	P.Ref `(Persona& p)`

Funciones que Retornan Tipos Compuestos

Por la misma razón y como *norma general*, salvo excepciones, tampoco es adecuado que una *función* retorne un valor de tipo compuesto, debido a la sobrecarga que generalmente ésto conlleva.

En estos casos, suele ser más adecuado que el subprograma devuelva el valor de tipo compuesto como un parámetro de salida mediante el paso por referencia.

6.2. Cadenas de Caracteres en C++: el Tipo `string`

En gran cantidad de ocasiones surge la necesidad de trabajar con datos representados por secuencias de caracteres. Pensemos, por ejemplo, en programas que manipulan nombres de personas, direcciones, etc. Por ese motivo, todos los lenguajes de programación consideran cómo procesar este tipo de situaciones, bien proporcionando tipos específicos para ello o proporcionando facilidades para que el programador defina tipos que permitan manipular cadenas de caracteres.

El lenguaje C++ dispone de la biblioteca estándar `<string>`, que proporciona el tipo `string` para representar cadenas de caracteres de *longitud finita*, limitada por la implementación. El tipo `string` dispone de operadores predefinidos que permiten manejar cadenas de caracteres de forma muy simple e intuitiva. También es posible manipular cadenas de caracteres con un estilo heredado del lenguaje C (como arrays de caracteres con terminador), pero el uso del tipo `string` permite definir cadenas de caracteres más robustas y con mejores características, por lo que en este curso siempre usaremos cadenas de caracteres de tipo `string`.

Una cadena de caracteres literal se representa mediante una sucesión de caracteres entre comillas dobles. Además, el tipo `string` puede ser utilizado para definir constantes simbólicas, variables o parámetros formales en los subprogramas. Así mismo, es posible asignar un valor de tipo `string` a una variable del mismo tipo, o utilizar los operadores relacionales (`==`, `!=`, `>`, `<`, `>=`, `<=`) para comparar cadenas de caracteres. Para utilizar el tipo `string` es necesario incluir la biblioteca estándar `<string>`, así como utilizar el espacio de nombres de `std`. El siguiente programa muestra algunos ejemplos:

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main()
{
    string nombre = "Pepe" ;
    // ...
    nombre = AUTOR ;
}
```

AUTOR:

J	o	s	e		L	u	i	s
0	1	2	3	4	5	6	7	8

nombre:

P	e	p	e
0	1	2	3

nombre:

J	o	s	e		L	u	i	s
0	1	2	3	4	5	6	7	8

Si la definición de una variable de tipo `string` no incluye la asignación de un valor inicial, dicha variable tendrá como valor por defecto la cadena vacía (`""`).

6.2.1. Entrada y Salida de Cadenas de Caracteres

La entrada/salida de datos de tipo `string` sigue el mismo esquema que la entrada/salida de los tipos predefinidos, explicada en el capítulo 3. Se basa en el uso de los operadores `>>` y `<<` sobre los flujos `cin` y `cout`. Su comportamiento es independiente del tipo de datos a leer, por lo que no haría falta incidir más en ello, sin embargo, a continuación presentamos algunos ejemplos que nos servirán de repaso, e introducimos algunas consideraciones específicas de la lectura datos de tipo `string`.

La utilización del operador `<<` sobre un flujo de salida `cout` muestra en la salida todos los caracteres que forman parte de la cadena. Por ejemplo, el siguiente código muestra en pantalla `"Nombre: José Luis"`:

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main()
{
```

```
    cout << "Nombre: " << AUTOR << endl ;
}
```

La utilización del operador >> sobre un flujo de entrada `cin` permite leer secuencias de caracteres y almacenarlas en variables de tipo `string`. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    string nombre ;
    cout << "Introduzca el nombre: " ;
    cin >> nombre ;
    cout << "Nombre: " << nombre << endl ;
}
```

Este operador de entrada (>>) se comporta (como se especificó en el capítulo 3 dedicado a la entrada y salida básica) de la siguiente forma: elimina los espacios en blanco que hubiera al principio de la entrada de datos, y lee dicha entrada hasta que encuentre algún carácter de espacio en blanco[1], que no será leído y permanecerá en el buffer de entrada (véase 3.1) hasta la próxima operación de entrada. En caso de que durante la entrada surja alguna situación de error, dicha entrada se detiene y el flujo de entrada se pondrá en un estado erróneo.

Nótese que, como consecuencia de lo anterior, no es posible utilizar el operador >> para leer una cadena de caracteres que incluya algún carácter en blanco. Por ejemplo, si como entrada al programa anterior introducimos por teclado la secuencia de caracteres *Juan Antonio*, veremos que en realidad la cadena leída es *Juan*, ya que el carácter en blanco actúa como delimitador y fuerza el fin de la lectura.

Si se desea leer una secuencia de caracteres que incluya espacios en blanco, utilizaremos la función `getline` en lugar del operador >>.

La función `getline` lee y almacena en una variable de tipo `string` todos los caracteres del *buffer* de entrada, hasta leer el carácter de fin de línea (ENTER), sin eliminar los espacios iniciales. Así pues, en nuestro ejemplo podríamos leer nombres compuestos utilizando `getline` de la siguiente forma:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    string nombre ;
    cout << "Introduzca el nombre: " ;
    getline(cin, nombre) ;
    cout << "Nombre: " << nombre << endl ;
}
```

Además, la función `getline` permite especificar el delimitador que marca el final de la secuencia de caracteres a leer. Si no se especifica ninguno (como ocurre en el ejemplo anterior), por defecto se utiliza el carácter de fin de línea. Sin embargo, si se especifica el delimitador, lee y almacena todos los caracteres del buffer hasta leer el carácter delimitador especificado, el cual es eliminado del buffer, pero no es almacenado en la variable. En el siguiente ejemplo se utiliza un punto como delimitador en `getline`, por lo que la lectura de teclado acaba cuanto se localice dicho carácter.

```
#include <iostream>
#include <string>
using namespace std ;
```

[1]Se consideran espacios en blanco los siguientes caracteres: espacio en blanco (`' '`), tabuladores (`'\t'`, `'\v'` y `'\f'`), retorno de carro (`'\r'`) y nueva línea (`'\n'`).

```
const char DELIMITADOR = '.' ;
int main()
{
    string nombre ;
    cout << "Introduzca el nombre: " ;
    getline(cin, nombre, DELIMITADOR) ;
    cout << "Nombre: " << nombre << endl ;
}
```

Como hemos visto, el comportamiento de las operaciones de lectura con `>>` y `getline` es diferente. En ocasiones, cuando se utiliza una lectura con `getline` después de una lectura previa con `>>`, podemos encontrarnos con un comportamiento que, aunque correcto, puede no corresponder al esperado intuitivamente. Conviene que conozcamos con detalle qué ocurre en cada caso, por lo que a continuación lo veremos sobre un ejemplo.

Supongamos que queremos diseñar un programa que, solicite y muestre el nombre y la edad de cinco personas. Para ello, leeremos el nombre de la persona con `getline` (ya que puede ser compuesto), y leeremos la edad con el operador de entrada `>>`, ya que nos permite introducir datos numéricos:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    string nombre ;
    int edad ;
    for (int i = 0; i < 5; ++i) {
        cout << "Introduzca el nombre: " ;
        getline(cin, nombre) ;
        cout << "Introduzca edad: " ;
        cin >> edad ;
        cout << "Edad: " << edad << " Nombre: [" << nombre << "]" << endl ;
    }
}
```

Sin embargo, al ejecutar el programa comprobamos que no funciona como esperábamos. La primera iteración funciona adecuadamente, el flujo de ejecución espera hasta que se introduce el nombre, y posteriormente espera hasta que se introduce la edad, mostrando dichos datos por pantalla. Sin embargo, las siguientes iteraciones funcionan de forma anómala, ya que la ejecución del programa no se detiene para que el usuario pueda introducir el nombre.

Esto es debido a que no hemos tenido en cuenta cómo se comportan las operaciones de lectura de datos (`>>` y `getline`) al obtener los datos del buffer de entrada. Hay que considerar que después de leer la edad en una determinada iteración, en el buffer permanece el carácter de fin de línea (ENTER) que se introdujo tras teclear la edad, ya que éste no es leído por el operador `>>`. En la siguiente iteración, la función `getline` lee una secuencia de caracteres hasta encontrar un ENTER (sin saltar los espacios iniciales), por lo que leerá el carácter ENTER que quedó en el buffer en la lectura previa de la edad de la iteración anterior, haciendo que finalice la lectura directamente. El resultado es que, al leer el nombre, se lee una cadena vacía, sin necesidad de detener el programa para que el usuario introduzca el nombre de la persona.

La solución a este problema es eliminar los caracteres de espacios en blanco (y fin de línea) del buffer de entrada. De esta forma el buffer estará realmente vacío y conseguiremos que la ejecución de `getline` haga que el programa se detenga hasta que el usuario introduzca el nombre. Hay diferentes formas de conseguir que el buffer se quede vacío.

Para eliminar los caracteres de espacios en blanco y fin de línea del buffer de entrada **antes** de leer la secuencia de caracteres con `getline`, utilizaremos el manipulador `ws` en el flujo `cin`, que extrae todos los espacios en blanco hasta encontrar algún carácter distinto, por lo que no será posible leer una cadena de caracteres vacía. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    string nombre ;
    int edad ;
    for (int i = 0; i < 5; ++i) {
        cin >> ws ;             // elimina los espacios en blanco y fin de línea
        cout << "Introduzca el nombre: " ;
        getline(cin, nombre) ;
        cout << "Introduzca edad: " ;
        cin >> edad ;
        cout << "Edad: " << edad << " Nombre: [" << nombre << "]" << endl ;
    }
}
```

También es posible que nos interese que la cadena vacía sea una entrada válida en el programa. En nuestro ejemplo podríamos estar interesados en que el usuario introduzca un nombre vacío como respuesta. En este caso, es necesario que el buffer se encuentre vacío en el momento de realizar la operación de entrada. Para ello, eliminaremos los caracteres que pudiera contener el buffer (no únicamente espacios en blanco) después de la última operación de lectura de datos, usando la función `ignore`. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    string nombre ;
    int edad ;
    for (int i = 0; i < 5; ++i) {
        cout << "Introduzca el nombre: " ;
        getline(cin, nombre) ;
        cout << "Introduzca edad: " ;
        cin >> edad ;
        cin.ignore(10000, '\n') ; // elimina todos los caracteres del buffer hasta '\n'
        cout << "Edad: " << edad << " Nombre: [" << nombre << "]" << endl ;
    }
}
```

La función `ignore` elimina todos los caracteres del buffer de entrada en el flujo especificado, hasta que se hayan eliminado el número de caracteres indicado en el primer argumento o bien se haya eliminado el carácter indicado en el segundo.

Nótese que la sentencia `cin >> ws` se asocia a la función `getline` que le sigue, mientras que la sentencia `ignore` se asocia a la sentencia de entrada >> que le precede.

6.2.2. Operaciones predefinidas

A continuación, usaremos algunos ejemplos para presentar las operaciones predefinidas más habituales sobre cadenas de tipo `string`.

- Asignación (=) de valores de tipo `string` a variables de tipo `string`:

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main()
```

```
{
    string nombre = "Pepe" ;
    // ...
    nombre = AUTOR ;
}
```

- Comparaciones lexicográficas[2] (==, !=, >, >=, <, <=):
  - `if (nombre >= AUTOR) { /*...*/ }`
- Concatenación de cadenas y de caracteres (+, +=):

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main ()
{
    string nombre = AUTOR + "López" ;
    nombre += "Vázque" ;
    nombre += 'z' ;
    nombre = AUTOR + 's' ;
}
```

- Paso como parámetro a subprogramas. Al igual que ocurre con los valores de otros tipos compuestos, podremos pasar cadenas como parámetros a subprogramas. Si el parámetro es de salida, o de entrada/salida, usaremos paso por referencia, y si es de entrada usaremos paso por referencia constante. Por ejemplo, si quisiéramos leer y escribir valores de tipo `string` podríamos declarar los subprogramas:

```
void leer_nombre(string& nm);            // parametro de salida
void escribir_nombre(const string& nm);  // parametro de entrada
```

- Cadenas como valor de retorno de funciones. Aunque está permitida la devolución de cadenas como valor de retorno de funciones, es una operación que está desaconsejada, debido a su alto coste. Por ello, cuando un subprograma necesite devolver una cadena, usualmente se devolverá como un parámetro de salida.
- Obtención del número de caracteres que componen la cadena (`size`):
  - `unsigned ncar = nombre.size();`
  - `if (nombre.size() == 0) { /*...*/ }`
- Acceso al i-ésimo carácter de la cadena (`[]`). El carácter accedido es de tipo `char`:
  - `char c = nombre[i];` donde i $\in$ `{ 0, ..., nombre.size()-1 }`
  - `nombre[i] = 'z';` donde i $\in$ `{ 0, ..., nombre.size()-1 }`

  Es importante tener en cuenta que el índice utilizado para acceder al carácter de la cadena debe corresponder a una posición válida de la misma. El acceso fuera del rango válido (por ejemplo, para añadir caracteres al final) es un error que hay que evitar. Aunque se trata de un error del programa, durante su ejecución no se nos avisa del mismo[3]. A partir de dicho momento el comportamiento del programa quedaría indeterminado. El tipo `string` dispone del operador `at` para acceder a posiciones de la cadena controlando posibles errores de acceso, pero no será utilizado es este curso.

---

[2] La comparación lexicográfica se basa en la ordenación alfabética, comúnmente utilizada en los diccionarios.

[3] En GNU C++ la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso de forma automática.

- Cambiar el número de caracteres contenidos en la cadena. Así, el método `nombre.resize(sz)` reajusta el número de caracteres contenidos en la cadena de caracteres `nombre` de tipo `string`.
  - Si el nuevo número de caracteres especificado (`sz`) es menor que el número actual de caracteres, entonces se eliminarán del final del *string* tantos caracteres como sea necesario para reducir su cantidad hasta el número de caracteres especificado.
  - Si por el contrario, el nuevo número de caracteres especificado (`sz`) es mayor que el número actual de caracteres, entonces se añadirán al final del *string* tantos caracteres como sea necesario hasta alcanzar el nuevo número de elementos especificado. Los nuevos caracteres introducidos tomarán, en caso de que exista, el valor especificado como segundo parámetro, o en otrocaso, tomarán un valor nulo.
  - Por ejemplo:
    - `string nombre = "pepe luis";` *nombre* contiene *pepe luis*.
    - `nombre.resize(4);` ahora *nombre* contiene *pepe*.
    - `nombre.resize(7, 'x');` ahora *nombre* contiene *pepexxx*.
- Obtención de una *nueva* subcadena (de tipo `string`) a partir del índice `i`. Se puede especificar el tamaño de la subcadena. Si no se especifica, o si el tamaño (`sz`) especificado excede al número de caracteres que hay desde `i`, entonces se toma la subcadena desde el índice hasta el final:
  - `string sb = nombre.substr(i);` donde `i` $\in$ `{ 0, ..., nombre.size()-1 }`
  - `string sb = nombre.substr(i, sz);` donde `i` $\in$ `{ 0, ..., nombre.size()-1 }`

  Nótese que **no** es válida la asignación a una subcadena: `nombre.substr(i, sz) = "...";`

6.2.3. Ejemplos

A continuación, mostramos algunos ejemplos de programas en los que trabajamos con cadenas de caracteres representadas mediante el tipo `string`.

Ejemplo 1. Conversión de una palabra a mayúsculas

Comenzamos con un programa que lee una palabra por teclado, la convierte a mayúsculas y la muestra en pantalla. Nótese que, aunque este programa es simple, no ha sido resuelto directamente en la función `main`. La lectura utiliza el operador `>>`, por lo que el programa únicamente puede recibir palabras que no incluyan espacios. El procedimiento `mayusculas` utiliza un parámetro de entrada/salida, por lo que se usa paso por referencia, como es habitual en cualquier tipo. La manipulación de la cadena, carácter a carácter, se realiza mediante un bucle típico en el que usamos `size` para conocer el número de caracteres de la cadena y el operador `[]` para acceder a un carácter concreto de la misma.

```
#include <iostream>
#include <string>
using namespace std ;
// -- Subalgoritmos ----
void mayuscula (char& letra)
{
    if ((letra >= 'a') && (letra <= 'z')) {
        letra = letra - 'a' + 'A' ;
    }
}
void mayusculas (string& palabra)
{
    for (unsigned i = 0 ; i < palabra.size() ; ++i) {
        mayuscula(palabra[i]) ;
```

```
    }
}
// -- Principal --------
int main ()
{
    string palabra ;
    cin >> palabra ;
    mayusculas(palabra) ;
    cout << palabra << endl ;
}
```

Nótese que un carácter concreto de la cadena es de tipo `char`, y se pasa por referencia a un subprograma que convierte una letra a su equivalente en mayúscula.

Ejemplo 2. Plural de una palabra

A continuación mostramos un programa que lee por teclado una palabra en minúsculas y muestra en pantalla su plural. El programa está diseñado en base a un procedimiento que usa un parámetro de entrada/salida en el que recibe la palabra original y devuelve su plural. Las reglas para convertir una palabra en plural son las siguientes:

- Si acaba en vocal se le añade la letra `'s'`.
- Si acaba en consonante se le añaden las letras `'es'`. Si la consonante es la letra `'z'`, se sustituye por la letra `'c'`.
- Suponemos que la palabra introducida es correcta y está formada por letras minúsculas.

Necesitamos acceder a la última letra de la cadena para determinar en qué caso nos encontramos, por lo que usamos `size` y el operador de acceso para consultar cuál es el último carácter. Una vez seleccionado el caso adecuado, procedemos a añadir caracteres a la palabra según corresponda. Usamos el procedimiento `plural_1` en el que se utiliza el operador de concatenación para añadir la terminación adecuada a la cadena. En caso de ser necesario, se accede a la última letra para cambiar la `'z'` por `'c'`. Nótese que para ello usamos el operador de acceso. Ello es posible porque dicha letra pertenece a la cadena y lo que queremos es sustituir un carácter existente por otro. Sin embargo, no es posible utilizar el operador de acceso para añadir la terminación `'s'` al final de la cadena, porque intentaríamos acceder a un carácter no existente en la misma, generando con ello un error.

Aunque desde el programa principal se hace uso del procedimiento `plural_1`, también se muestra una implementación alternativa en el procedimiento `plural_2`. En este caso nos basamos en la posibilidad de utilizar `substr` para obtener una subcadena. Cuando es necesario, tomamos la cadena excluyendo la letra final y al resultado le concatenamos la terminación `"ces"`.

```
#include <iostream>
#include <string>
using namespace std ;
// -- Subalgoritmos ----
bool es_vocal (char c)
{
    return (c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') || (c == 'u') ;
}
void plural_1 (string& palabra)
{
    if (palabra.size() > 0) {
        if (es_vocal(palabra[palabra.size() - 1])) {
            palabra += 's' ;
        } else {
            if (palabra[palabra.size() - 1] == 'z') {
```

```
                palabra[palabra.size() - 1] = 'c' ;
            }
            palabra += "es" ;
        }
    }
}
void plural_2 (string& palabra)
{
    if (palabra.size() > 0) {
        if (es_vocal(palabra[palabra.size() - 1])) {
            palabra += 's' ;
        } else if (palabra[palabra.size() - 1] == 'z') {
            palabra = palabra.substr(0, palabra.size() - 1) + "ces" ;
        } else {
            palabra += "es" ;
        }
    }
}
// -- Principal --------
int main ()
{
    string palabra ;
    cin >> palabra ;
    plural_1(palabra) ;
    cout << palabra << endl ;
}
```

Ejemplo 3. Función palíndromo

Se dice que una cadena es un palíndromo si se puede leer igual de derecha a izquierda o de izquierda a derecha, por ejemplo, *ana*. Queremos implementar una función que reciba como parámetro de entrada una cadena con una palabra y devuelva si se trata de un palíndromo. Como el parámetro de entrada es de tipo `string` (un tipo compuesto), utilizamos paso por referencia constante. La implementación se basa en considerar que, para que la cadena sea palíndromo, deben coincidir dos a dos cada par de letras situadas en posiciones simétricas respecto al carácter situado en la posición central de la cadena. Usamos un bucle para comparar cada par de letras y dos índices para hacer referencia, respectivamente, al carácter a la izquierda y a la derecha de la cadena. Si el bucle termina después de comparar todos los posibles pares, entonces se trata de un palíndromo.

```
bool es_palindromo (const string& palabra)
{
    bool ok = false ;
    if (palabra.size() > 0) {
        unsigned i = 0 ;
        unsigned j = palabra.size() - 1 ;
        while ((i < j) && (palabra[i] == palabra[j])) {
            ++i ;
            --j ;
        }
        ok = i >= j ;
    }
    return ok ;
}
```

Ejemplo 4. Sustituir una subcadena por otra

Queremos disponer de un subprograma que reciba una cadena y reemplace la subcadena que empieza en una cierta posición `i` y con un tamaño `sz` por una nueva cadena `nueva`. Por ejemplo,

al reemplazar en la cadena "camarero" la subcadena que comienza en la posición 2 y tiene 3 caracteres por la cadena sill, la cadena se convierte en "casillero".

El siguiente subprograma reemplazar define el parámetro de entrada/salida str (por referencia), los parámetros de entrada i y sz (por valor, al ser de tipos simples) y el parámetro de entrada nueva (por referencia constante, al ser de tipo string). Su implementación se basa en usar subsr para obtener las subcadenas adecuadas y el operador de concatenación para formar la cadena resultante adecuadamente.

```
void reemplazar (string& str, unsigned i, unsigned sz, const string& nueva)
{
    if (i + sz < str.size()) {
        str = str.substr(0, i) + nueva + str.substr(i + sz, str.size() - (i + sz)) ;
    } else if (i <= str.size()) {
        str = str.substr(0, i) + nueva ;
    }
}
```

Nota: La biblioteca <string> contiene la función replace, que podría haber sido utilizada directamente para obtener el objetivo propuesto. Este subprograma es equivalente a la operación str.replace(i, sz, nueva).

6.3. Registros o Estructuras

Un *registro* representa un valor compuesto por un número determinado de elementos, que pueden ser de distintos tipos (simples y compuestos). Para utilizar registros definiremos un **nuevo tipo** *registro*, enumerando los elementos (*campos*) que lo componen. Para cada campo deberemos indicar su tipo y el identificador con el nos referiremos al mismo.

La definición del tipo registro se hará utilizando la palabra reservada struct, seguido del identificador con el que haremos referencia a dicho tipo. A continuación se enumeran, entre llaves, los campos que lo componen, especificando su tipo y el identificador con el que referenciarlo, seguido por el delimitador punto y coma (;). La llave de cierre debe ir seguida de punto y coma.

A continuación mostramos la definición de un tipo para representar fechas, dadas por tres números que representan un cierto día de un mes de un año.

```
struct Fecha {
    unsigned dia ;
    unsigned mes ;
    unsigned anyo ;
} ;
```

Una vez definido el tipo registro, podrá ser utilizado como cualquier otro tipo, para definir constantes simbólicas, variables o parámetros formales en los subprogramas. Por ejemplo, podemos utilizar el nuevo tipo Fecha para definir variables:

```
Fecha f_ingreso ;
```

o para definir constantes:

```
const Fecha F_NAC = { 20, 2, 2001 } ;
```

Los valores de tipo estructurado están formados por diferentes componentes, por lo que necesitamos alguna notación especial para indicar claramente el valor de cada una de ellos. Como se puede observar en el ejemplo, el valor que queremos que tome la constante F_NAC se expresa enumerando y separando por comas los valores que queremos asignar a los campos (en el mismo orden de la definición del tipo registro) y utilizando llaves para agruparlo todo.

Los valores del tipo Fecha se componen de tres elementos concretos, el día (de tipo unsigned), el mes (de tipo unsigned) y el año (de tipo unsigned). Los identificadores dia, mes y anyo

representan los nombres de sus elementos componentes, denominados **campos**, cuyo ámbito de visibilidad se restringe a la propia definición del registro. Los campos de un registro pueden ser de cualquier tipo de datos, simple o compuesto. Por ejemplo, podríamos estar interesados en tratar con información de empleados, definiendo un nuevo tipo `Empleado` como un registro que contiene el nombre del empleado (de tipo `string`), su código y sueldo (de tipo `unsigned`) y su fecha de ingreso en la empresa (de tipo `Fecha`).

```
// -- Tipos ------------
struct Empleado {
    string   nombre ;
    unsigned codigo ;
    unsigned sueldo ;
    Fecha    fecha_ingreso ;
} ;
// -- Principal --------
int main ()
{
    Empleado e ;
    // ...
}
```

Una vez declarada una entidad (constante o variable) de tipo registro, por ejemplo la variable `f_ingreso`, podemos referirnos a ella en su globalidad (realizando asignaciones y pasos de parámetros) o acceder a sus componentes (campos) utilizando el operador punto (`.`). Una vez que accedemos a un determinado campo tenemos un valor del tipo de dicho campo, por lo que podrá ser utilizado de acuerdo a las características de dicho tipo, como si se tratase de una variable de dicho tipo.

```
int main ()
{
    Fecha f_nac, hoy ;
    hoy.dia = 18 ;
    hoy.mes = 10 ;
    hoy.anyo = 2001 ;

    f_nac = hoy ;
}
```

	f_ nac		*hoy*
dia:	18	dia:	18
mes:	10	mes:	10
anyo:	2001	anyo:	2001

6.3.1. Operaciones con registros completos

Además de utilizar el operador punto para acceder individualmente a campos concretos de un registro, es posible efectuar operaciones con un registro completo, sin considerar campos concretos. Únicamente se permiten tres tipos de operaciones con registros completos:

- Asignación.

  Es posible utilizar el operador de asignación para asignar un valor de tipo registro a una variable del mismo tipo registro. Por ejemplo, si `f1` y `f2` son dos variables de tipo `Fecha`, podríamos hacer lo siguiente para almacenar cada uno de los campos de `f2` en el correspondiente campo de `f1`.

  ```
  f1 = f2 ;
  ```

  cuyo efecto es equivalente a la copia uno a uno de los campos, aunque obviamente expresado de forma más legible, intuitiva y menos propensa a errores.

  ```
  f1.dia = f2.dia ;
  f1.mes = f2.mes ;
  f1.anyo = f2.anyo ;
  ```

En general, es posible asignar un valor de tipo registro completo a una variable o campo, siempre que sea del mismo tipo. Por ejemplo, podríamos asignar un registro de tipo `Fecha` al campo `fecha_ingreso` de un registro de tipo `Empleado`, ya que tanto el valor que se asigna como el elemento al que se asigna son del mismo tipo.

```
Empleado e;
Fecha f2 = { 18, 10, 2001 } ;

e.nombre = "Juan" ;
e.codigo = 101 ;
e.sueldo = 1000 ;
e.fecha_ingreso = f2;
```

- Paso como parámetro a subprogramas. Al igual que ocurre con los valores de otros tipos compuestos, podremos pasar registros como parámetros a subprogramas. Si el parámetro es de salida, o de entrada/salida, usaremos paso por referencia, y si es de entrada usaremos paso por referencia constante. Por ejemplo, si quisiéramos leer y escribir valores de tipo `Fecha` podríamos declarar los subprogramas:

```
void leer_fecha(Fecha& f);            // parametro de salida
void escribir_fecha(const Fecha& f);  // parametro de entrada
```

- Registros como valor de retorno de funciones. Aunque está permitida la devolución de registros como valor de retorno de funciones, es una operación que está desaconsejada, debido a su alto coste. Por ello, cuando un subprograma necesite devolver un registro, usualmente se devolverá como un parámetro de salida.

No hay ninguna otra operación disponible con registros completos. Sin embargo, esto no constituye ninguna limitación, porque el programador puede definir subprogramas que reciban registros como parámetros. De esta forma puede disponer de operaciones que hagan más simple el diseño del programa[4].

6.3.2. Entrada/Salida de valores de tipo registro

Un tipo registro es definido por el programador, por lo que no existe un mecanismo predefinido para la lectura o escritura de valores de tipo registro. El programador deberá ocuparse de la lectura/escritura de un registro, efectuando la lectura/escritura de cada uno de sus campos. Es recomendable ocultar todos estos detalles por lo que, en general, definiremos subprogramas para leer o escribir valores de cada uno de los tipos registros que se utilicen en un programa. Por ejemplo, en un programa que utilizara el tipo `Fecha`, sería recomendable definir los siguientes subprogramas:

```
void leer_fecha (Fecha& f)
{
    cin >> f.dia >> f.mes >> f.anyo ;
}
void escribir_fecha (const Fecha& f)
{
    cout << f.dia << "/" << f.mes << "/" << f.anyo ;
}
```

6.3.3. Ejemplo. Uso de registros

A continuación presentamos un programa en el que trabajamos con información que representa instantes de tiempo, dados por tres números que representan cierta hora, minuto y segundo. El programa lee de teclado dos valores de tiempo y muestra en pantalla la diferencia entre ambos.

[4] Nótese que los operadores típicos de comparacián (`==`, `!=`, `>`, `<`, ...) no son válidos entre valores de tipo registro.

Podríamos abordar este programa sin necesidad de definir tipos registros, utilizando múltiples variables para representar cada uno de los elementos manipulados. Por ejemplo, podríamos definir las variables `d1`, `h1` y `m1` para representar la hora, minuto y segundo del primer instante. Procederíamos igual con el segundo, con la diferencia, etc. El resultado sería un programa con multitud de variables con las que hacer referencia a conceptos que, en realidad, están relacionados. Por este motivo, resulta más adecuado definir un nuevo tipo `Tiempo` como un registro con tres campos, representando la hora, los minutos y los segundos. De esta forma, para representar un instante dado bastará con un único valor de tipo `Tiempo`, compuesto de tres campos. Manejamos la misma información, pero de una forma más organizada, legible y compacta.

El siguiente programa muestra esta segunda solución. Como se puede ver, la definición del tipo `Tiempo` nos permite usar únicamente tres variables en el programa principal, que corresponden con los conceptos manejados en el mismo: el primer tiempo, el segundo y su diferencia. Así mismo, la descomposición modular del programa, definiendo subprogramas para leer, escribir convertir valores de tiempo a segundos o calcular la diferencia de dos instantes de tiempo, permite que el programa principal quede muy legible y resulte intuitivo.

```
#include <iostream>
#include <string>
using namespace std ;
// -- Constantes -------
const unsigned SEGMIN = 60 ;
const unsigned MINHOR = 60 ;
const unsigned MAXHOR = 24 ;
const unsigned SEGHOR = SEGMIN * MINHOR ;
// -- Tipos ------------
struct Tiempo {
    unsigned horas ;
    unsigned minutos ;
    unsigned segundos ;
} ;
// -- Subalgoritmos ----
unsigned leer_rango (unsigned inf, unsigned sup)
{
    unsigned num ;
    do {
        cin >> num ;
    } while ( ! ((num >= inf) && (num < sup))) ;
    return num ;
}
void leer_tiempo (Tiempo& t)
{
    t.horas = leer_rango(0, MAXHOR) ;
    t.minutos = leer_rango(0, MINHOR) ;
    t.segundos = leer_rango(0, SEGMIN) ;
}
void escribir_tiempo (const Tiempo& t)
{
    cout << t.horas << ":" << t.minutos << ":" << t.segundos ;
}
unsigned tiempo_a_seg (const Tiempo& t)
{
    return (t.horas * SEGHOR) + (t.minutos * SEGMIN) + (t.segundos) ;
}
void seg_a_tiempo (unsigned sg, Tiempo& t)
{
    t.horas = sg / SEGHOR ;
    t.minutos = (sg % SEGHOR) / SEGMIN ;
    t.segundos = (sg % SEGHOR) % SEGMIN ;
```

```
}
void diferencia (const Tiempo& t1, const Tiempo& t2, Tiempo& dif)
{
    seg_a_tiempo(tiempo_a_seg(t2) - tiempo_a_seg(t1), dif) ;
}
// -- Principal --------
int main ()
{
    Tiempo t1, t2, dif ;
    leer_tiempo(t1) ;
    leer_tiempo(t2) ;
    diferencia(t1, t2, dif) ;
    escribir_tiempo(dif) ;
    cout << endl ;
}
```

6.4. Agregados: el Tipo Array

Un *array* representa un valor compuesto por un número determinado (definido en tiempo de compilación) de elementos de un *mismo tipo* de datos. Este tipo de valores son útiles en aquellas situaciones en las que necesitamos almacenar y manipular una colección de valores y acceder a ellos de forma parametrizada, normalmente para aplicar un proceso iterativo. Por ejemplo, podríamos estar interesados en almacenar notas de un examen de los 100 alumnos de un curso y, posteriormente, calcular la media de aquellos que aprueban.

El programador definirá un nuevo tipo *agregado* cuando decida que necesita manipular valores con las características mencionadas. Para ello, nos basaremos en el tipo **array** de la biblioteca estándar de C++[5], por lo que habrá que incluir la biblioteca <array>.

Un tipo *agregado* se define utilizando typedef (indicando así que estamos definiendo un nuevo tipo de datos), seguido de una descripción de las características de los elementos de dicho tipo y de un nombre con el que identificarlo. Las características del nuevo tipo se describen utilizando **array** e indicando el tipo de sus elementos (su tipo *base*) y el número de elementos, que **debe ser** una constante conocida en tiempo de compilación. El resultado es un array con el número de elementos y el tipo especificado, en el que cada elemento está identificado por un valor numérico que va desde cero (primer elemento) hasta el número de elementos menos uno (último elemento). Por ejemplo, podemos definir un nuevo tipo Vector como un agregado de 5 elementos, cada uno del tipo int:

```
#include <array>
using namespace std ;
// -- Constantes -------
const int NELMS = 5 ;
// -- Tipos ------------
typedef array<int, NELMS> Vector ;
```

Posteriormente podremos usar dicho tipo Vector para definir variables y constantes como es usual. Sin embargo, como ahora tratamos con valores compuestos, las constantes literales del tipo array se especifican entre *llaves dobles*. Por ejemplo, a continuación definimos una constante PRIMOS con los primeros números primos, y una variable v, cuyo valor inicial está sin especificar.

```
// -- Constantes -------
const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }} ;
// -- Principal --------
int main ()
{
    Vector v;
}
```

PRIMOS:

0	1	2	3	4
2	3	5	7	11

v:

0	1	2	3	4
?	?	?	?	?

[5] El tipo array de la biblioteca estándar está disponible desde el estándar C++11. Existen otras formas de trabajar con arrays, aunque no las estudiaremos en este curso.

El tipo base (de los elementos) del array puede ser simple o compuesto. Por ejemplo, podemos definir un nuevo tipo `Citas` como un agregado de 4 elementos, cada uno del tipo `Fecha`, y definir variables y constantes de dicho tipo:

```
struct Fecha {
    unsigned dia;
    unsigned mes;
    unsigned anyo;
};
const int N_CITAS = 4;
typedef array<Fecha, N_CITAS> Citas ;
const Citas CUMPLEANYOS = {{
    { 1, 1, 2001 },
    { 2, 2, 2002 },
    { 3, 3, 2003 },
    { 4, 4, 2004 }
}} ;
int main()
{
    Citas cit;
}
```

CUMPLEANYOS:

0	1	2	3
1	2	3	4
1	2	3	4
2001	2002	2003	2004

Al igual que cuando trabajamos con registros nos interesa acceder a sus campos para manipular sus valores adecuadamente, al trabajar con arrays nos interesará acceder a sus componentes individuales. Para ello, en este caso se utiliza el operador (`[]`), indicando dentro de los corchetes el índice de la posición que ocupa el elemento al que nos referimos. Por ejemplo, con `cit[0]` accedemos a un componente de tipo `Fecha` situado en la primera posición del array `cit`. Una vez que accedemos a un elemento del array, éste puede ser utilizado exactamente igual que un valor del tipo base del mismo. En nuestro ejemplo, una vez que accedemos a `cit[0]`, lo que tenemos es un registro de tipo `Fecha`, por lo que podremos manipularlo exactamente igual que si se tratara de una variable de dicho tipo. Así, si quisiera establecer que el día almacenado en la fecha situada en el primera componente del array sea el 22, bastaría con hacer lo siguiente:

```
cit[0].dia = 22 ;
```

El programador es responsable de hacer un uso adecuado de los elementos del array, accediendo a posiciones válidas del mismo. Para ello, deberá tener en cuenta que el índice del primer elemento del array es 0 y el índice del último elemento viene dado por el número de elementos con que se ha definido **menos uno**. Dicho número de elementos es conocido por ser el valor de la constante utilizada en el `typedef` (en nuestro ejemplo `N_CITAS`) , aunque resulta más adecuado utilizarlo accediendo a la función `size()` sobre la variable de tipo array correspondiente (en nuestro ejemplo, `cit.size()`).

Si por error se intentara acceder a una posición no válida de un array, se estaría generando una situación anómala. No se produciría ningún aviso de dicho error y, a partir de ese momento, el programa podría tener un comportamiento inesperado. Por ejemplo, el siguiente programa define una variable de tipo `Citas` en la que almacena unas determinadas fechas. Sin embargo, al salir del bucle accede a una posición errónea.

```
struct Fecha {
    unsigned dia, mes, anyo;
};
const int N_CITAS = 4;
typedef array<Fecha, N_CITAS> Citas ;
int main()
{
    Citas cit;
    cit[0].dia = 18;
    cit[0].mes = 10;
```

```
    cit[0].anyo = 2001;
    for (int i = 0; i < cit.size(); ++i) {
        cit[i].dia = 1;
        cit[i].mes = 1;
        cit[i].anyo = 2002;
    }
    cit[N_CITAS] = { 1, 1, 2002 };    // ERROR. Acceso fuera de los limites
    // ...
}
```

Si utilizamos el compilador GNU C++, es posible compilar con la opción de compilación `-D_GLIBCXX_DEBUG`, que permite comprobar los índices al acceder a un array[6]

También es posible acceder a un determinado elemento del array mediante la operación `at()`, la cual controla posibles errores de acceso, pero no será utilizada en este curso.

6.4.1. Operaciones predefinidas

Los tipos arrays definidos en base al tipo array de la biblioteca estándar de C++ se pueden manipular como el resto de valores de otros tipos de datos, por lo que se pueden realizar de forma predefinida las siguientes operaciones:

- Asignación (=) de valores de un tipo array a una variable del mismo tipo array. Por ejemplo, dado el tipo `Vector` introducido anteriormente, podríamos hacer:

```
Vector v1, v2;

...
v1 = v2;
```

- Comparación de igualdad (`==`). Se obtiene `true` o `false` según coincida o no cada elemento del primer array con el elemento correspondiente (en la misma posición) del segundo array. Es aplicable si el operador `==` está definido para elementos del tipo base.

- Comparación de desigualdad (`!=`). Se obtiene `true` si algún elemento del primer array no coincide con el elemento correspondiente (en la misma posición) del segundo. Es aplicable si el operador `!=` está definido para elementos del tipo base.

- Comparaciones lexicográficas (`>`, `<`, `>=`, `<=`). Se obtiene `true` si el primer operando satisface la operación especificada respecto al segundo. Es aplicable si el operador relacional está definido para elementos del tipo base.

- Paso como parámetro a subprogramas. Al igual que ocurre con los valores de otros tipos compuestos, podremos pasar arrays como parámetros a subprogramas. Si el parámetro es de salida, o de entrada/salida, usaremos paso por referencia, y si es de entrada usaremos paso por referencia constante. Por ejemplo, si quisiéramos leer y escribir valores de tipo `Citas` podríamos declarar los subprogramas:

```
void leer_citas(Citas& c);             // parametro de salida
void escribir_citas(const Citas& c);  // parametro de entrada
```

- Arrays como valor de retorno de funciones. Aunque está permitida la devolución de arrays como valor de retorno de funciones, es una operación que está desaconsejada, debido a su alto coste. Por ello, cuando un subprograma necesite devolver un array, usualmente se devolverá como un parámetro de salida.

- Entrada/salida. Al igual que ocurre en el caso de los tipos registro y, en general, para cualquier tipo de datos definido por el usuario, no es posible disponer de operaciones predefinidas en C++ para su entrada/salida. El programador deberá ocuparse de efectuar la entrada/salida de datos de tipo array, leyendo o escribiendo cada componente según el tipo de que se trate.

[6]Si el alumno desea utilizar esta opción, debe descargar la biblioteca de la página web de la asignatura

6.4.2. Ejemplos

A continuación, mostraremos la utilidad práctica de la definición de tipos arrays, mediante ejemplos cuya solución sería poco factible si únicamente dispusiéramos de los tipos vistos hasta ahora.

Ejemplo 1. Vectores

Supongamos que queremos diseñar un programa que trabaje con vectores de 5 elementos. Sabemos que utilizaremos algunas operaciones típicas sobre los vectores como leer todas las componentes desde teclado, imprimir el vector, calcular el producto escalar de dos vectores, calcular su suma, etc. Podríamos haber pensado en declarar 5 variables individuales para representar las componentes de cada vector, (por ejemplo, v11, v12, v13, v14 y v15), pero resulta evidente que este enfoque nos conduce a una solución inaceptable. Nos llevaría a un programa con multitud de variables y casi imposible de manejar ¿Qué hacemos si el vector tiene 100 elementos?, ¿cómo abordamos posibles cambios futuros?. En este ejemplo la única solución factible es definir un nuevo tipo que esté compuesto por 5 elementos del mismo tipo base y sobre el que podamos iterar para acceder sucesivamente a sus diferentes componentes.

A continuación definimos el tipo `Vector` y procesamos sus elementos mediante bucles. Ello nos permite *recorrer* los elementos, visitando uno a uno cada elemento, para efectuar la operación adecuada en cada caso. Por ejemplo, la lectura del vector se basa en un bucle en el que usamos una variable de control y hacemos que tome el valor que nos interesa para determinar la posición del elemento del array en la que almacenar el valor leído.

```
#include <iostream>
#include <array>
using namespace std;
// -- Constantes -------
const unsigned NELMS = 5;
// -- Tipos ------------
typedef array<int, NELMS> Vector;
// -- Subalgoritmos ----
void leer (Vector& v)
{
    for (unsigned i = 0; i < v.size(); ++i) {
        cin >> v[i];
    }
}
int sumar (const Vector& v)
{
    int suma = 0;
    for (unsigned i = 0; i < v.size(); ++i) {
        suma += v[i];
    }
    return suma;
}
// -- Principal --------
int main ()
{
    Vector v1, v2;
    leer(v1);
    leer(v2);
    if (sumar(v1) == sumar(v2)) {
        cout << "Misma suma" << endl;
    }
}
```

Ejemplo 2. Cálculo del sueldo de los agentes en una empresa

Queremos implementar un programa que calcule e imprima en pantalla el sueldo de los 20 agentes de ventas de una empresa. Cada agente cobra un sueldo fijo de 1000 €más un incentivo, que será un 10 % de las ventas que ha realizado. Sin embargo, dicho incentivo no será percibido por todos los agentes, sino sólo por aquellos cuyas ventas superen los 2/3 de la media de ventas del total de los agentes. El planteamiento del problema nos obliga a almacenar las ventas de todos los agentes, porque no es posible decidir si un agente cobra o no el incentivo hasta que se hayan leído las ventas de todos los agentes y calculado el umbral que determina si un agente cobra o no su incentivo.

De nuevo, debemos excluir la posibilidad de usar variables individuales para las ventas de cada agente, y optaremos por organizar las ventas de todos los agentes definiendo un nuevo tipo compuesto en el que cada componente represente las ventas de un determinado agente. En el siguiente programa definimos el tipo `Ventas` y los subprogramas adecuados para procesar valores de dicho tipo. Usamos el esquema típico para el paso de parámetros, según se trate de parámetros de entrada o de salida y utilizamos los bucles adecuados para recorrer cada array, accediendo y procesando sus componentes una a una.

```
#include <iostream>
#include <array>
using namespace std;
// -- Constantes -------
const unsigned NAGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// -- Tipos ------------
typedef array<double, NAGENTES> Ventas;
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (unsigned i = 0; i < v.size(); ++i) {
        suma += v[i];
    }
    return suma / double(v.size());
}
double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0;
}
void leer_ventas (Ventas& v)
{
    for (unsigned i = 0; i < v.size(); ++i) {
        cout << "Introduzca ventas del Agente " << i << ": ";
        cin >> v[i];
    }
}
void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(v);
    for (unsigned i = 0; i < v.size(); ++i) {
        double sueldo = SUELDO_FIJO;
        if (v[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v[i]);
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
    }
}
```

```
// -- Principal --------
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}
```

6.4.3. Listas o Secuencias con Número Variable de Elementos. Agregados Incompletos

Hay situaciones en las que no se conoce exactamente la cantidad de elementos que deseamos almacenar en un array. Por otra parte, sabemos que el tipo array requiere que el número de elementos del mismo sea fijo y conocido en tiempo de compilación (antes de ejecutar el programa). Como consecuencia, no es posible hacer que el tamaño del array se adapte exactamente al tamaño necesitado durante la ejecución. En estos casos optaremos por seleccionar un tipo array con un número de elementos que permita almacenar el mayor número de elementos previsto en el programa. En estas situaciones nos encontramos con que el tamaño del array no coincide con el número de elementos almacenados en el mismo. Pensemos, por ejemplo, en un programa que gestione una agenda de contactos como el que presentamos en la sección 6.5. En este caso, al principio no hay ningún contacto almacenado y, a medida que se añaden o eliminan contactos, su número va variando. La agenda se define como un array con un número fijo de elementos, pero al principio del programa no hay ninguno y durante su ejecución vamos añadiendo y eliminando contactos según decida el usuario.

Necesitamos alguna forma de determinar claramente qué elementos del array contienen los datos reales del programa. Por ejemplo, si en nuestra agenda tenemos tres contactos, ¿cuales son?: ¿los tres primeros?, ¿los tres últimos?, ¿el primero y los dos últimos?. La solución a esta situación es programar una estrategia para almacenar los elementos en el array y, posteriormente, usar la misma estrategia para localizarlos.

Se pueden seguir dos enfoques: almacenar los elementos contiguos en el array o bien permitir que estén dispersos, con posibles *huecos* entre ellos. Por lo general, gestionar el array con huecos durante la ejecución del programa suele ser complejo e ineficiente. Así pues, salvo que se justifique lo contrario, optaremos por mantener los elementos válidos del array almacenados en posiciones consecutivas del mismo. En este caso, necesitamos algún criterio para determinar en qué posición del array acaban los elementos válidos y comienzan las posiciones que no nos interesan. Hay dos posibilidades:

- Definir un elemento reconocible del array que nos permita localizar el punto de frontera entre ambas zonas.
- Contabilizar el número de elementos válidos almacenados en el array.

La primera opción suele requerir la localización del elemento que delimita la frontera entre los elementos válidos y los no utilizados. Por este motivo, esta opción suele ser, en la mayoría de los casos, más compleja e ineficiente. Nosotros optaremos por seguir la segunda alternativa. En este caso, deberemos plantearnos cómo conocer el número de elementos válidos del array. De nuevo, ahora se plantean dos opciones: mantener dicho número independientemente del array (en una variable adicional), o bien asociarlo al array al que se refiere, definiendo un registro que contenga dos campos: el array con los elementos almacenados y el número de elementos válidos del mismo.

En general, optaremos por esta segunda posibilidad, ya que ello da lugar a programas con mejores características y no introduce complejidad adicional. Este enfoque únicamente requiere efectuar la correspondiente definición de tipos en base a un registro y el acceso a los elementos del array y al número total de elementos almacenados en el mismo, sabiendo que se trata de campos de un determinado valor de tipo registro. Por ejemplo, en el programa de gestión de sueldos de los agentes de ventas, podríamos considerar un número variable de agentes (con un máximo de 20). De esta forma, ahora definiríamos el tipo `Ventas` de la siguiente forma:

```
#include <iostream>
#include <array>
using namespace std;
// -- Constantes -------
const unsigned MAX_AGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// -- Tipos ------------
typedef array<double, MAX_AGENTES> Datos;
struct Ventas {
    unsigned nelms;
    Datos elm;
};
```

Como se puede observar, los subprogramas `calc_media` e `imprimir_sueldos` son prácticamente idénticos a los presentados en la versión anterior salvo que, como el array es un campo de un registro, ahora usamos una notación diferente para acceder a sus componentes. De forma similar, ahora el número de agentes no es un valor fijo, sabemos que está almacenado en el campo correspondiente del mismo registro.

```
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (unsigned i = 0; i < v.nelms; ++i) {
        suma += v.elm[i];
    }
    return suma / double(v.nelms);
}
double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0;
}
void imprimir_sueldos (const Ventas& v) {
    double umbral = PROMEDIO * calc_media(v);
    for (unsigned i = 0; i < v.nelms; ++i) {
        double sueldo = SUELDO_FIJO;
        if (v.elm[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v.elm[i]);
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
    }
}
```

En este problema debemos optar por un criterio para leer los elementos de la entrada. En la versión anterior bastaba con leer 20 números, porque sabíamos que siempre había 20 agentes. Ahora el número de elementos a introducir puede ser diferente y deberemos decidir cómo queremos que tenga lugar la lectura de datos. En el programa mostramos dos de las opciones más frecuentes para este tipo de casos:

- Que el usuario introduzca datos hasta teclear un valor que indique el fin del proceso de lectura. En nuestro caso, (en el subprograma `leer_ventas_1`) detectamos el fin del proceso de lectura cuando, o bien se han introducido las ventas del número máximo de agentes, o bien se introduce un dato de ventas incorrecto (es cero o menor).

```
// ----------------------------------
void leer_ventas_1 (Ventas& v) {
    double vent_ag;
```

```
    v.nelms = 0;
    cout << "Introduzca ventas del agente " << v.nelms + 1 << ": ";
    cin >> vent_ag;
    while ((v.nelms < v.size())&&(vent_ag > 0)) {
        v.elm[v.nelms] = vent_ag;
        ++v.nelms;
        cout << "Introduzca ventas del agente " << v.nelms + 1 << ": ";
        cin >> vent_ag;
    }
}
```

- Que el usuario comunique por adelantado el número de datos de agentes a leer en total. Esta opción resulta más simple, porque se puede programar con el mismo esquema usado para un número fijo de agentes. Como se puede observar en el subprograma `leer_ventas_2`, la única diferencia con el programa para un número fijo de agentes es que ahora el número de elementos a leer no viene dado por un valor fijo, sino por el valor leído al principio de la secuencia de entrada. Nótese cómo, para asegurar que el programa no intenta trabajar con más datos de los previstos, al leer el número de agentes comprobamos que no sea erróneo, avisando con un mensaje de error adecuado en caso de ser necesario.

```
// ----------------------------------
void leer_ventas_2 (Ventas& v) {
    unsigned nag;
    cout << "Introduzca total de agentes: ";
    cin >> nag;
    if (nag > v.size()) {
        v.nelms = 0;
        cout << "Error" << endl;
    } else {
        v.nelms = nag;
        for (unsigned i = 0; i < v.nelms; ++i) {
            cout << "Introduzca ventas del agente " << v.nelms + 1 << ": ";
            cin >> v.elm[i];
        }
    }
}
// -- Principal --------
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}
```

6.4.4. Agregados Multidimensionales

Hasta ahora hemos tratado con arrays de una dimensión. En este caso podemos hacernos la idea de que el array es como un casillero en el que podemos utilizar un índice para acceder a una determinada posición. Ello se corresponde con la idea de una estructura lineal, con *una dimensión* en el espacio. Sin embargo, sabemos que el *tipo base* de un array puede ser tanto simple como compuesto, incluyendo la posibilidad de que sea un array. De esta forma, podemos tener *arrays con múltiples dimensiones*. Si cada elemento de un array es otro array, ello puede ser considerado como un array bidimensional, coincidiendo con la idea espacial de superficie o con la idea de tabla. En este caso, necesitaríamos dos índices para acceder a cada elemento individual del array (el primero para hacer referencia a la fila y el segundo para la columna). De igual forma, si este elemento es a su vez un array, obtendríamos un array de tres dimensiones, lo que coincide con la idea espacial de volumen, necesitando tres índices para acceder a cada elemento individual. Podríamos continuar de igual forma, aunque ya no disponemos de una idea espacial que lo represente el valor definido.

En el siguiente ejemplo se define el tipo `Matriz`, que representa arrays de dos dimensiones cuyos elementos básicos son de tipo `int`. Para ello, definimos el tipo `Fila` como un array unidimensional de `int`, y el tipo `Matriz` como un array de filas.

```
#include <iostream>
#include <array>
using namespace std;
// -- Constantes -------
const unsigned NFILAS = 3;
const unsigned NCOLUMNAS = 5;
// -- Tipos ------------
typedef array<int, NCOLUMNAS> Fila ;
typedef array<Fila, NFILAS> Matriz ;
// -- Principal --------
int main ()
{
    Matriz m;
    for (unsigned f = 0; f < m.size(); ++f) {
        for (unsigned c = 0; c < m[f].size(); ++c) {
            m[f][c] = (f * 10) + c;
        }
    }
}
```

`m:`

0	00	01	02	03	04
1	10	11	12	13	14
2	20	21	22	23	24
	0	1	2	3	4

Una vez definida una variable `m` de tipo `Matriz`, su procesamiento puede requerir trabajar con una fila completa, en cuyo caso utilizaríamos un único índice. Por ejemplo `m[0]` hace referencia a la componente de índice 0 de la matriz `m` que, según la definición, es de tipo `Fila`. Así mismo, podemos estar interesados en procesar un elemento concreto de tipo `int`, en cuyo caso necesitamos dos índices. Por ejemplo, `m[0][2]` hace referencia a la componente de índice 2 dentro de la fila 0, que es de tipo `int`.

Del mismo modo, `m.size()` representa el número de filas de la matriz `m`, y `m[f].size()` representa el número de elementos de la fila `f` de la matriz `m`.

Ejemplo. Procesamiento básico de arrays bidimensionales

A continuación presentamos un ejemplo sencillo de procesamiento de arrays bidimensionales. Pretendemos leer de teclado números correspondientes a una matriz de 3×5, efectuar algunas operaciones con la matriz y mostrar en pantalla los resultados, de acuerdo a un determinado formato. Usaremos un array bidimensional para almacenar los número leídos y haremos una lectura suponiendo que los números son introducidos fila a fila. Finalmente, imprimiremos la matriz según el siguiente formato:

a	a	a	a	a	b
a	a	a	a	a	b
a	a	a	a	a	b
c	c	c	c	c	

donde *a* representa los elementos de la matriz leída desde el teclado, *b* representa el resultado de sumar todos los elementos de la fila correspondiente, y *c* representa el resultado de sumar todos los elementos de la columna en que se encuentran.

El programa utiliza la definición del tipo `Matriz` introducido anteriormente, y utiliza diversos subprogramas para efectuar cada una de las operaciones requeridas con la matriz.

```
#include <iostream>
using namespace std;
// -- Constantes -------
const unsigned NFILAS = 3;
const unsigned NCOLUMNAS = 5;
// -- Tipos ------------
```

```
typedef array<int, NCOLUMNAS> Fila ;
typedef array<Fila, NFILAS> Matriz ;
```

La función `sumar_fila` recibe como parámetro una fila de la matriz y calcula la suma de sus elementos. Aunque el programa principal manipula una matriz bidimensional, el valor utilizado como parámetro actual en la llamada (`m[f]`) es un elemento de la matriz que, al ser de tipo `Fila`, encaja en la definición del parámetro formal correspondiente. Como se puede apreciar, dentro del subprograma trabajamos con `fil` que es de tipo `Fila`, por lo que para acceder a los números a sumar utilizamos un único índice. En realidad, `sumar_fila` procesa un array de una dimensión, independientemente de que la llamada sea una fila de una matriz, o simplemente un vector.

```
// -- Subalgoritmos ----
int sumar_fila (const Fila& fil)
{
    int suma = 0;
    for (unsigned c = 0; c < fil.size(); ++c) {
        suma += fil[c];
    }
    return suma;
}
```

La función `sumar_columna` no puede recibir como parámetro una única *columna*. El programa no contiene la definición de ningún tipo que corresponda con lo que nosotros entendemos por una columna. Ello es algo que nos imaginamos al pensar en la matriz, pero que no está definido en el programa. Por tanto, la definición de `sumar_columna` necesita toda la información necesaria para acceder a los elementos de una determinada columna. Es decir, la matriz completa y un número que indica la columna cuyos elementos queremos sumar. Como se puede apreciar, dentro del subprograma trabajamos con `m` que es de tipo `Matriz`, por lo que para acceder a los números a sumar utilizamos dos índices.

```
int sumar_columna (const Matriz& m, unsigned c)
{
    int suma = 0;
    for (unsigned f = 0; f < m.size(); ++f) {
        suma += m[f][c];
    }
    return suma;
}
```

Al igual que ocurre con `sumar_fila`, para escribir una fila de la matriz podemos aprovechar que existe un tipo `Fila`, por lo que este subprograma no es más que la escritura en pantalla de un array de una dimensión.

```
void escribir_fila (const Fila& fil)
{
    for (unsigned c = 0; c < fil.size(); ++c) {
        cout << fil[c] << " ";
    }
}
```

Además, como hemos definido numerosos subprogramas de apoyo, vemos que el procesamiento del array bidimensional completo queda reducido a un recorrido típico de un array, accediendo a una fila completa (`m[f]`) cada vez que queremos procesar una fila para escribirla en pantalla o para calcular su suma.

```
void escribir_matriz_formato (const Matriz& m)
{
    for (unsigned f = 0; f < m.size(); ++f) {
```

```
        escribir_fila(m[f]);
        cout << sumar_fila(m[f]);
        cout << endl;
    }
    for (unsigned c = 0; c < m[0].size(); ++c) {
        cout << sumar_columna(m, c) << " ";
    }
    cout << endl;
}
```

Finalmente, aunque también podríamos haber definido un subprograma `leer_fila` para leer una fila, e implementar la operación `leer_matriz`, haciendo llamadas a dicha operación, a continuación mostramos otro posible enfoque. Ahora implementamos la lectura de elementos fila a fila y su almacenamiento en la matriz mediante dos bucles anidados. De esta forma, ahora utilizamos dos índices para acceder a la casilla en la que almacenar el número leído.

```
void leer_matriz (Matriz& m)
{
    cout << "Escribe fila a fila" << endl;
    for (unsigned f = 0; f < m.size(); ++f) {
        for (unsigned c = 0; c < m[0].size(); ++c) {
            cin >> m[f][c];
        }
    }
}
// -- Principal --------
int main ()
{
    Matriz m;
    leer_matriz(m);
    escribir_matriz_formato(m);
}
```

6.5. Resolución de Problemas Utilizando Tipos Compuestos. Agenda

Finalizaremos el tema resolviendo un problema completo en el que integramos los elementos fundamentales tratados en el capítulo. Para ello, diseñaremos un programa para gestionar una agenda personal en la que almacenaremos el nombre, el teléfono y la dirección de un número variable de personas. El programa ofrecerá un menú con las opciones típicas para manipular una agenda:

- Añadir los datos de una persona.
- Acceder a los datos de una persona a partir de su nombre.
- Borrar una persona a partir de su nombre.
- Modificar los datos de una persona a partir de su nombre.
- Listar el contenido completo de la agenda.

Necesitamos definir un tipo array en el que almacenar la información de las personas de la agenda. Dicha definición requiere conocer el número máximo de elementos del array, por lo que asumiremos que nuestra agenda no contendrá más de 50 contactos. Este problema encaja dentro de lo que hemos denominado *tratamiento de arrays incompletos*, porque el número de posiciones realmente utilizadas del array puede variar y no coincide con el número total de elementos del array. Al principio no habrá ningún contacto y, conforme se vayan añadiendo y eliminado contactos, el número de posiciones ocupadas irá variando. Por este motivo, definiremos un tipo `Agenda` como un

registro con dos campos: uno de ellos contiene el número actual de contactos almacenados y otro el array con la información detallada de los mismos.

De cada contacto almacenamos su nombre, su dirección y su teléfono, por lo que conviene agrupar todos estos elementos en un tipo común. Para ello, definimos el tipo `Persona` como un registro con tres campos: su nombre y su teléfono, de tipo `string`, y su dirección. Aunque se podría haber optado por definir el teléfono como un campo de tipo `unsigned`, hemos preferido hacerlo de tipo `string` porque no pensamos manipularlo con operaciones aritméticas sino con operaciones de cadenas (por ejemplo, podríamos pensar en obtener la subcadena que determina el prefijo de la provincia). La dirección de una persona vendrá dada por una calle, un piso, un código postal y una ciudad. Por ese motivo, definiremos un nuevo tipo registro, llamado `Direccion`, que lo represente. Nótese que estamos trabajando con un registro (de tipo `Persona`) en el que, a su vez, uno de sus campos es otro registro (de tipo `Direccion`).

```
#include <iostream>
#include <string>
#include <cassert>
#include <array>
using namespace std ;

// -- Constantes -------
const unsigned MAX_PERSONAS = 50 ;
// -- Tipos ------------
struct Direccion {
    unsigned num ;
    string calle ;
    string piso ;
    string cp ;
    string ciudad ;
} ;
struct Persona {
    string nombre ;
    string tel ;
    Direccion direccion ;
} ;
typedef array<Persona, MAX_PERSONAS> Personas ;
struct Agenda {
    unsigned n_pers ;
    Personas pers ;
} ;
```

Se ha definido el tipo enumerado `Cod_Error` para definir valores que representen las posibles situaciones de error en el programa. Como veremos más adelante, usaremos valores dicho tipo para determinar si una operación se ha realizado correctamente (`OK`) o por el contrario se ha producido alguna situación de error al ejecutar el programa.

```
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
} ;
```

Hemos definido una serie de subprogramas que nos permiten descomponer el programa en operaciones independientes. Utilizamos el subprograma `inicializar` para obtener una agenda que esté vacía (es decir, que no contenga ningún elemento). Dado que hemos seguido el criterio de organizar los elementos de la agenda situándolos contiguos y al principio, y de usar un campo con el número total de elementos, nos bastará con hacer que el campo `n_pers` tome el valor cero. Nótese que, aunque parezca que esta operación tiene poca entidad como para separarla en un subprograma independiente, en realidad ocurre justamente lo contrario. Se trata de una operación típica a realizar con la agenda, que debe ser tratada de forma independiente. Ello permite que, por

ejemplo, si en un futuro deseamos cambiar la implementación de la agenda, los cambios puedan ser fácilmente localizados y realizados de forma segura.

También hemos utilizado subprogramas que permitan abordar de forma fácil y aislada la entrada/salida de los diferentes tipos definidos por el usuario. Ello suele ser una buena práctica en general, y es especialmente adecuado en este problema. Como resultado, disponemos de operaciones para `leer_direccion`, `escribir_direccion`, `leer_persona` y `escribir_persona`.

```
// -- Subalgoritmos ----
void inicializar (Agenda& ag)
{
    ag.n_pers = 0 ;
}
void leer_direccion(Direccion& dir)
{
    cin >> dir.calle ;
    cin >> dir.num ;
    cin >> dir.piso ;
    cin >> dir.cp ;
    cin >> dir.ciudad ;
}
void escribir_direccion(const Direccion& dir)
{
    cout << dir.calle << " " ;
    cout << dir.num << " " ;
    cout << dir.piso << " " ;
    cout << dir.cp << " " ;
    cout << dir.ciudad << " " ;
}
void leer_persona(Persona& per)
{
    cin >> per.nombre ;
    cin >> per.tel ;
    leer_direccion(per.direccion) ;
}
void escribir_persona(const Persona& per)
{
    cout << per.nombre << " " ;
    cout << per.tel << " " ;
    escribir_direccion(per.direccion) ;
    cout << endl ;
}
```

Al tratar con colecciones de datos surge frecuentemente la necesidad de localizar un dato determinado para procesarlo de alguna forma. Nosotros tratamos las colecciones de datos como parte de agregados o arrays, por lo que una operación para *buscar* en qué posición del array se encuentra un determinado elemento, resulta especialmente útil.

Utilizamos un función `buscar_persona` que, dada una agenda y el nombre de una persona, nos indica en qué posición se encuentra, o bien que no se encuentra. Para ello, supondremos que si el valor devuelto corresponde a una posición del array con un dato válido ello indica que se encuentra en dicha posición, mientras que si corresponde a una posición no válida es porque no se encuentra.

```
unsigned buscar_persona(const string& nombre, const Agenda& ag)
{
    unsigned i = 0 ;
    while ((i < ag.n_pers) && (nombre != ag.pers[i].nombre)) {
        ++i ;
    }
    return i ;
}
```

Utilizamos los subprograma `anyadir_persona`, `borrar_persona` y `modificar_persona` para proporcionar las operaciones básicas con las que se añade, elimina o modifica información de la agenda. Estos subprogramas se apoyan en `anyadir` y `eliminar`, que se encargan específicamente de incluir un elemento nuevo en la agenda y de borrar el elemento de una posición dada. La implementación de la agenda no requiere que los elementos se encuentren en un orden concreto, por lo que el elemento a añadir se situará al final de la lista de elementos válidos. Así mismo, para eliminar el elemento situado en una cierta posición únicamente debemos garantizar que los elementos válidos restantes están situados de forma consecutiva, por lo que trasladaremos el elemento situado al final a la posición del hueco generado en la posición del elemento borrado.

Algunos subprogramas devuelven un parámetro `ok`, de salida, que se utiliza para considerar posibles situaciones de error. Por ejemplo, podría ocurrir que el usuario deseara borrar una persona que no se encuentra en la agenda. Para detectar y reaccionar ante esta situación, dichos subprogramas devuelven un valor de tipo `Cod_Error` que indica la posible situación de error.

En el ejemplo citado, la operación `borrar_persona` devuelve el valor `OK` si la operación se pudo ejecutar correctamente y el valor `NO_ENCONTRADO` si no se pudo ejecutar (porque la persona no se encontraba almacenada en la agenda). Una vez que la función principal dispone del valor que informa si se ha producido un error, éste es usado en la llamada a `escribir_cod_error`. Dicho subprograma es invocado cada vez que el usuario utiliza el menú para ejecutar una operación de la agenda. Se encarga de informar de si la operación se pudo ejecutar con normalidad o, por el contrario, se produjo alguna situación de error.

```
void anyadir(Agenda& ag, const Persona& per)
{
    ag.pers[ag.n_pers] = per ;
    ++ag.n_pers ;
}
void eliminar (Agenda& ag, unsigned pos)
{
    if (pos < ag.npers-1) {
        ag.pers[pos] = ag.pers[ag.n_pers - 1] ;
    }
    --ag.n_pers ;
}
void anyadir_persona(const Persona& per, Agenda& ag, Cod_Error& ok)
{
    unsigned i = buscar_persona(per.nombre, ag) ;
    if (i < ag.n_pers) {
        ok = YA_EXISTE ;
    } else if (ag.n_pers >= ag.pers.size()) {
        ok = AG_LLENA ;
    } else {
        ok = OK ;
        anyadir(ag, per) ;
    }
}
void borrar_persona(const string& nombre, Agenda& ag, Cod_Error& ok)
{
    unsigned i = buscar_persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        eliminar(ag, i) ;
    }
}
void modificar_persona(const string& nombre, const Persona& nuevo, Agenda& ag,
                       Cod_Error& ok)
{
```

```
    unsigned i = buscar_persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        eliminar(ag, i) ;
        anyadir_persona(nuevo, ag, ok) ;
    }
}
```

Los subprogramas `imprimir_persona` e `imprimir_agenda` presentan en pantalla información solicitada por el usuario. Nótese que, aunque en `imprimir_agenda` no es posible que se produzca ninguna situación de error, también se utiliza un parámetro `ok`, en el que siempre se devuelve `OK`. Ello permite utilizar la operación `imprimir_agenda` desde el menú principal homogéneamente con el resto de operaciones del usuario.

```
void imprimir_persona(const string& nombre, const Agenda& ag, Cod_Error& ok)
{
    unsigned i = buscar_persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        escribir_persona(ag.pers[i]) ;
    }
}
void imprimir_agenda(const Agenda& ag, Cod_Error& ok)
{
    for (unsigned i = 0 ; i < ag.n_pers ; ++i) {
        escribir_persona(ag.pers[i]) ;
    }
    ok = OK ;
}
```

Los subprogramas `menu` y `escribir_cod_error` se encargan de la interacción con el usuario, bien para seleccionar la opción adecuada o para presentar el estado resultante de ejecutar cada opción.

```
char menu ()
{
    char opcion ;
    cout << endl ;
    cout << "a. - Añadir Persona" << endl ;
    cout << "b. - Buscar Persona" << endl ;
    cout << "c. - Borrar Persona" << endl ;
    cout << "d. - Modificar Persona" << endl ;
    cout << "e. - Imprimir Agenda" << endl ;
    cout << "x. - Salir" << endl ;
    do {
        cout << "Introduzca Opción: " ;
        cin >> opcion ;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x'))) ;
    return opcion ;
}
void escribir_cod_error (Cod_Error cod)
{
    switch (cod) {
    case OK:
        cout << "Operación correcta" << endl ;
        break ;
```

```
        case AG_LLENA:
            cout << "Agenda llena" << endl ;
            break ;
        case NO_ENCONTRADO:
            cout << "La persona no se encuentra en la agenda" << endl ;
            break ;
        case YA_EXISTE:
            cout << "La persona ya se encuentra en la agenda" << endl ;
            break ;
        }
    }
```

El programa principal se reduce a la utilización de los subprogramas definidos previamente. Nótese que, dado que la manipulación de la agenda se hace en el subprograma que corresponda, la implementación concreta de la agenda está oculta para el programa principal, por lo que queda claramente legible y fácilmente modificable.

```
    // -- Principal --------
    int main ()
    {
        Agenda ag ;
        char opcion ;
        Persona per ;
        string nombre ;
        Cod_Error ok ;
        inicializar(ag) ;
        do {
            opcion = menu() ;
            switch (opcion) {
            case 'a':
                cout << "Introduzca los datos de la Persona" << endl ;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
                leer_Persona(per) ;
                anyadir_persona(per, ag, ok) ;
                escribir_cod_error(ok) ;
                break ;
            case 'b':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                imprimir_persona(nombre, ag, ok) ;
                escribir_cod_error(ok) ;
                break ;
            case 'c':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                borrar_persona(nombre, ag, ok) ;
                escribir_cod_error(ok) ;
                break ;
            case 'd':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                cout << "Nuevos datos de la Persona" << endl ;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
                leer_persona(per) ;
                modificar_persona(nombre, per, ag, ok) ;
                escribir_cod_error(ok) ;
                break ;
            case 'e':
                imprimir_agenda(ag, ok) ;
```

```
                escribir_cod_error(ok) ;
                break ;
            }
        } while (opcion != 'x' ) ;
    }
```

Capítulo 7

Búsqueda y Ordenación

Al desarrollar programas es frecuente que nos encontremos con situaciones en las que es necesario acceder a (*buscar*) un determinado elemento de una colección con un objetivo concreto. Podemos estar interesados, por ejemplo, en mostrarlo en pantalla, en eliminarlo o en modificarlo según resulte conveniente. En el capítulo anterior ya nos percatamos de ello al desarrollar nuestro programa para gestionar una agenda de contactos. La gran importancia de este tipo de operaciones de búsqueda hace que nos planteemos su estudio de forma más detallada. A continuación mostraremos algunas posibilidades para abordar la búsqueda. Hay otras posibilidades, como el uso de técnicas de búsqueda en tablas hash que, aunque son de gran utilidad, no trateremos en este capítulo.

Hasta ahora hemos tratado con colecciones de elementos almacenadas en arrays, pero no nos hemos preocupado de la disposición interna de dichos elementos en el array. Lo importante ha sido que los elementos se encuentren almacenados en el array y que podamos acceder a ellos, pero sin imponer un criterio de almacenamiento concreto. Sin embargo, hay determinadas ocasiones en los que podemos estar interesados en que los elementos se encuentren organizados de acuerdo a un determinado criterio. Por ejemplo, podríamos estar interesados en que los elementos se almacenen en la agenda ordenados ascendentemente por el nombre de la persona. En este capítulo revisaremos algunas estrategias típicas para *ordenar* los elementos en un array.

A continuación, asumiremos que la colección de elementos con la que trabajamos (tanto para búsqueda como para ordenación) se encuentra almacenada en un array, para lo que usaremos el siguiente tipo `Vector`.

```
//--------------------------------
const unsigned MAXIMO = 50;
typedef array<int, MAXIMO> Vector ;
```

7.1. Algoritmos de Búsqueda

En general, si deseamos localizar un elemento en un array es porque después queremos hacer algo con dicho elemento. Por este motivo, diseñaremos nuestros algoritmos de búsqueda de forma que devuelvan la posición del elemento que queremos localizar, o bien una indicación de que el elemento buscado no se se encuentra en la colección.

```
//--------------------------------
// busca la posición del primer elemento igual a x
// si no se encuentra, retorna v.size()
//-------------
unsigned buscar(int x, const Vector& v);
```

Hay diferentes criterios para reconocer que un elemento no se encuentra en la colección. En general, basta con obtener una posición *no válida*. Nosotros supondremos que esta posición no

válida es el índice de un elemento del array mayor que el último elemento de la colección (usaremos v.size()).

7.1.1. Búsqueda Lineal (Secuencial)

Consiste en ir recorriendo secuencialmente la colección de datos hasta encontrar el elemento buscado o, en última instancia, hasta recorrer la colección completa, en cuyo caso podemos estar seguros de que el elemento buscado no se encuentra en la colección.

La búsqueda lineal es adecuada como mecanismo de búsqueda general en colecciones de datos *sin organización* conocida. A continuación se muestra el algoritmo básico, que puede ser adaptado según las circunstancias.

```
//--------------------------------
// busca la posición del primer elemento igual a x
// si no se encuentra, retorna v.size()
//-------------
unsigned buscar(int x, const Vector& v)
{
    unsigned i = 0 ;
    while ((i < v.size())&&(x != v[i])) {
        ++i ;
    }
    return i ;
}
//--------------------------------
```

Otra posible implementación alternativa es la siguiente.

```
//--------------------------------
int buscar_alt(int x, const Vector& v)
{
    int idx = int(v.size());
    bool ok = false;
    for (int i = 0; (i < int(v.size())) && ! ok; ++i) {
        if (x == v[i]) {
            ok = true;
            idx = i;
        }
    }
    return idx;
}
//--------------------------------
```

Como puede observarse, recorremos uno a uno todos los elementos hasta que podemos responder en sentido afirmativo o negativo. Respondemos en sentido negativo (el elemento no se encuentra) si el índice del siguiente elemento a probar está más allá del último elemento del array. Respondemos en sentido positivo si el elemento indicado por la variable `i` contiene el elemento buscado. En tal caso, acaba el bucle y se devuelve dicha posición `i`. Nótese que si el elemento no se encuentra se devuelve `v.size()`, que es una posición no válida del array.

Un programa que haga uso de la función `buscar` usará el valor devuelto para determinar si el elemento a buscar se encuentra o no en el array. Por ejemplo, el siguiente fragmento elimina un elemento del array en caso de que se encuentre y controla las situaciones de error que se puedan dar.

```
    unsigned p = buscar(num, v) ;
    if (p < v.size()){
        eliminar(v, p) ;
        cod_err = OK ;
    }else{
```

```
        cod_err = NO_ENCONTRADO ;
    }
```

El acceso a elementos de un array exige estar seguro de que no accedemos a elementos fuera de los índices definidos para el mismo. Por este motivo, es importante que el orden en el que se evalúan las diferentes condiciones en la expresión de control del fin del bucle sea el mostrado en el algoritmo. De esa forma, aprovechamos la evaluación en *cortocircuito* y garantizamos que el bucle se detiene cuando no hay más elementos a inspeccionar (`i >= v.size()`), evitando así accesos erróneos a posiciones no válidas del array. Si hubiéramos escrito el bucle permutando las dos partes de la expresión de control del bucle:

```
    while ((x != v[i])&&(i < v.size())) {
```

tendríamos un algoritmo *incorrecto*, porque si el elemento no se encuentra, intentaríamos acceder posiciones del array que no forman parte del mismo.

Búsqueda Lineal Multidimensional 2D

Al igual que la búsqueda lineal considerada anteriormente, la búsqueda lineal multidimensional 2D consiste en ir recorriendo secuencialmente la colección de datos (en este caso estructurada en dos dimensiones) hasta encontrar el elemento buscado o, en última instancia, hasta recorrer la colección completa, en cuyo caso podemos estar seguros de que el elemento buscado no se encuentra en la colección.

```
const unsigned NCOLS = 5 ;
const unsigned NFILS = 7 ;
typedef array<int, NCOLS> Fila ;
typedef array<Fila, NFILS> Matriz ;
//--------------------------------
// busca la posición del primer elemento igual a x
// si no se encuentra, f tomará el valor v.size()
// y c un valor inespecificado
//-------------
void buscar(int x, const Matriz& m, unsigned& f, unsigned& c)
{
    f = 0 ;
    c = 0 ;
    while ((f < m.size())&&(x != m[f][c])) {
        ++c ;
        if (c >= m[f].size()) {
            c = 0 ;
            ++f ;
        }
    }
}
//--------------------------------
```

Otra posible implementación alternativa es la siguiente.

```
//--------------------------------
void buscar_alt(int x, const Matriz& m, int& ff, int& cc)
{
    ff = int(m.size());
    bool ok = false;
    for (int f = 0; (f < int(m.size())) && ! ok; ++f) {
        for (int c = 0; (c < int(m[f].size())) && ! ok; ++c) {
            if (x == m[f][c]) {
                ok = true;
                ff = f;
```

```
                cc = c;
            }
        }
    }
}
//--------------------------------
```

7.1.2. Búsqueda Binaria

La búsqueda secuencial es muy simple, pero requiere recorrer todos los elementos del array para estar seguros de que el elemento a buscar no se encuentra en la colección. Si el array tiene gran cantidad de elementos y necesitamos que las búsquedas se realicen de forma rápida, este mecanismo podría no ser adecuado.

Si estamos interesados en acelerar el proceso de búsqueda, necesitamos información adicional que nos de *pistas* para poder realizar una implementación más eficiente. Necesitamos tener información acerca de la organización interna de los elementos en el array.

Hay diferentes formas de organizar la información en el array que nos permiten implemementar algoritmos de búsqueda más eficientes. A continuación estudiaremos el algoritmo de búsqueda binaria, que asume que las colecciones de datos se encuentran almacenadas en el array de forma *ordenadas* según algún criterio.

La idea consiste en seleccionar un elemento de la colección y comprobar si se trata del elemento buscado. Si es así el proceso termina con éxito, pero si no lo es, podemos aprovechar que sabemos que los elementos se encuentran ordenados y descartar *todos* los elementos que se encuentran a la derecha del mismo o a su izquierda (según la relación entre el valor seleccionado y el valor buscado). Este proceso se repite hasta encontrar el elemento o hasta que no queden elementos en la colección, en cuyo caso el elemento no habrá sido encontrado. A continuación se presenta el algoritmo básico, que puede ser adaptado según el contexto concreto en el que se quiera utilizar.

```
//--------------------------------
// busca la posición del primer elemento igual a x
// si no se encuentra, retorna v.size()
//-------------
unsigned buscar_bin(int x, const Vector& v)
{
    unsigned i = 0 ;
    unsigned f = v.size() ;
    unsigned m = (i + f) / 2 ;
    while ((i < f) && (x != v[m])) {
        if (x < v[m]) {
            f = m ;
        } else {
            i = m + 1 ;
        }
        m = (i + f) / 2 ;
    }
    if (i >= f) {
        m = int(v.size());
    }
    return m;
}
//--------------------------------
```

Como se puede observar, utilizamos dos índices (`i` y `f`) para delimitar la zona del array con elementos entre los que buscar. El índice `i` (inicio) indica el primer elemento válido del array y el índice `f` (fin) indica el primer elemento no válido. Al principio, la zona coincide con el array completo, por lo que hacemos que `i` tome el valor 0 y `f` tome el valor `v.size()` (el primero no válido). Mientras queden elementos entre los que buscar (`i < f`), seleccionamos uno con el que probar. Lo más óptimo es seleccionar el central (`(i + f) / 2`), porque de esa forma descartamos

un mayor número de elementos, en caso de que el elemento seleccionado no sea el buscado. En cada paso probamos con el elemento seleccionado y actualizamos los índices adecuadamente como consecuencia de la comparación entre el elemento seleccionado y el buscado.

Salimos del bucle si encontramos el elemento en la posición `m` o si los índices `i` y `f` se cruzan (indicando que no quedan elementos entre los que seguir buscando). Al salir del bucle nos aseguramos que la variable `m` contenga el valor adecuado, es decir, o bien un valor fuera del rango válido (`v.size()`), en caso de que no se encuentre el elemento, o bien la posición del elemento encontrado.

7.2. Algoritmos de ordenación

El problema de cómo ordenar los elementos en un array aparece frecuentemente en la bibliografía básica de programación, bien como base para proponer algoritmos típicos sobre los que estudiar propiedades de los programas o como objeto de estudio para obtener algoritmos eficientes que reduzcan el coste computacional del proceso de ordenación. Nosotros utilizamos este problema como parte de nuestro aprendizaje de programación básica, por lo que presentaremos algunos de los algoritmos más típicos, sin preocuparnos de buscar soluciones especialmente eficientes, sino buscando soluciones claras, simples y fáciles de entender. Para encontrar propuestas más eficientes debe consultarse bibliografía más avanzada.

7.2.1. Ordenación por Selección

La idea de este algoritmo consiste en buscar el menor elemento de aquellos a ordenar y situarlo en su posición (al principio). De esta forma, la colección a ordenar pasa a tener un elemento menos, y basta con repetir el proceso, pero considerando una colección desordenada menor. El proceso termina cuando la colección a ordenar contiene un único elemento.

El siguiente subprograma `seleccion` muestra nuestra solución, que describe exactamente la idea presentada anteriormente. Se basa en el uso de un subprograma `subir_menor`, que considera un array sin ordenar y sitúa el menor elemento al principio del mismo. Haciendo uso de este subprograma, basta con realizar sucesivas iteraciones en las que usamos `subir_menor` identificando los elementos del array que quedan sin ordenar. Para ello, el subprograma `subir_menor` recibe, además del array a ordenar, el índice (`pos`) que delimita la parte del mismo que está aún sin ordenar.

```
//--------------------------------
void seleccion(Vector& v)
{
    for (unsigned pos = 0 ; pos < v.size()-1 ; ++pos) {
        subir_menor(v, pos) ;
    }
}
//--------------------------------
```

El subprograma `subir_menor` recibe una zona de un array, marcada desde un cierto elemento `pos` hasta el final, y sitúa el menor elemento al principio. Si es necesario, el elemento que ocupaba la primera posición es situado en la posición que ocupaba el menor. Necesitamos localizar la posición del menor elemento del array, por lo que utilizamos un subprograma `posicion_menor`.

```
//--------------------------------
inline void subir_menor(Vector& v, unsigned pos)
{
    unsigned pos_menor = posicion_menor(v, pos) ;
    if (pos != pos_menor) {
        intercambio(v[pos], v[pos_menor]) ;
    }
}
```

Finalmente, solo queda proceder a la implementación de los correspondientes subprogramas para localizar el menor elemento de un array y para intercambiar dos elementos.

```
//--------------------------------
inline void intercambio(int& x, int& y)
{
    int a = x ;
    x = y ;
    y = a ;
}
//--------------------------------
unsigned posicion_menor(const Vector& v, unsigned pos)
{
    int pos_menor = pos ;
    for (unsigned i = pos_menor+1 ; i < v.size() ; ++i) {
        if (v[i] < v[pos_menor]) {
            pos_menor = i ;
        }
    }
    return pos_menor ;
}
```

7.2.2. Ordenación por Intercambio (Burbuja)

La idea de este algoritmo consiste en hacer repetidas pasadas sobre el array, trasladando en cada una el elemento más pequeño hasta el principio del array. Este algoritmo se conoce como ordenación por el método de la *burbuja*, porque si se consideran los elementos como si estuviera en posición vertical y fueran burbujas con un cierto *peso* en un depósito de agua, dichas burbujas irían ascendiendo en función de su valor.

Como se puede apreciar, al igual que en el algoritmo de ordenación por selección, también tenemos sucesivas pasadas en las que conseguimos situar un elemento al principio. La diferencia está en el mecanismo utilizado para conseguirlo. Por ello, el subprograma `burbuja` sigue el mismo esquema que el subprograma `seleccion` presentado anteriormente, salvo que ahora la implementación del subprograma `subir_menor` es diferente, respondiendo a la idea de burbuja.

```
//--------------------------------
void subir_menor(Vector& v, unsigned pos)
{
    for (unsigned i = v.size()-1 ; i > pos ; --i) {
        if (v[i] < v[i-1]) {
            intercambio(v[i], v[i-1]) ;
        }
    }
}
//--------------------------------
void burbuja(Vector& v)
{
    for (int pos = 0 ; pos < v.size()-1 ; ++pos) {
        subir_menor(v, pos) ;
    }
}
//--------------------------------
```

7.2.3. Ordenación por Inserción

En este caso, la idea consiste en cosiderar dos zonas en el array, la primera contiene elementos ordenados y la segunda contiene elementos desordenados. En cada paso se toma un elemento de la zona desordenada y se inserta en la zona de elementos ordenados, de forma que quede en la

posición adecuada. Conseguimos que la zona ordenada tenga un elemento más y la desordenada un elemento menos. El proceso continúa hasta que la zona desordenada no contiene ningún elemento.

En nuestra implementación usamos `pos` para delimitar el comienzo de la zona de elementos desordenados. Todos los elementos a su izquierda estarán ordenados y en cada paso se insertará el elementos indicado por `pos` en la zona ordenada del array. Nótese cómo hacemos que `pos` tome inicialmente el valor 1. Ello representa la situación inicial, en la que la zona ordenada contiene un único elemento y la desordenada el resto.

En cada iteración localizamos la posición en la que debemos situar el elemento tratado (`v[pos]`) y procedemos a su inserción. Si al elemento a insertar le corresponde ir al final de la zona ordenada, ya se encuentra en su posición correcta, por lo que no habría que hacer nada. Sin embargo, en cualquier otro caso deberemos garantizar que, una vez situado el elemento en la posición adecuada, todos los elementos sigan estando ordenados. Para ello, antes de almacenar el elemento en la posición destino abrimos un hueco en dicha posición, desplazando cada elemento una posición a su derecha.

```
//--------------------------------
void insercion(Vector& v)
{
    for (unsigned pos = 1 ; pos < v.size() ; ++pos) {
        unsigned p_hueco = buscar_posicion(v, pos) ;
        if (p_hueco != pos) {
            int aux = v[pos] ;
            abrir_hueco(v, p_hueco, pos) ;
            v[p_hueco] = aux ;
        }
    }
}
//--------------------------------
```

Ls búsqueda de la posición del elemento es un recorrido de búsqueda normal, en el que comprobamos la condición en la zona del array delimitada entre `0` y `pos`. Como puede observarse, en este caso sabemos que `v[pos]` no cumplirá la condicion, por lo que no es necesario utilizar una condición compuesta (no es necesario comprobar que se alcanza el fin de la zona de búsqueda). A continuación se muestra el algoritmo, en el que omitimos (comentamos) dicha parte de la comprobación.

```
//--------------------------------
unsigned buscar_posicion(const Vector& v, unsigned pos) {
    unsigned i = 0 ;
    while (/*(i < pos)&&*/ (v[pos] > v[i])) {
        ++i ;
    }
    return i ;
}
```

El subprograma `abrir_hueco` desplaza uno a uno los elementos en una zona dada, avanzando de derecha a izquierda para evitar que el desplazamiento de un elemento afecte al siguiente.

```
//--------------------------------
void abrir_hueco(Vector& v, unsigned p_hueco, unsigned p_elm)
{
    for (unsigned i = p_elm ; i > p_hueco ; --i) {
        v[i] = v[i-1] ;
    }
}
```

7.3. Aplicación de los Algoritmos de Búsqueda y Ordenación

Finalmente, adaptaremos nuestra solución al problema de la agenda personal. Ahora consideraremos que queremos que la información se encuentre internamente almacenada de forma ordenada. De esta forma, podremos proceder a efectuar búsquedas mediante el algoritmo de búsqueda binaria, consiguiendo así que el proceso de búsqueda sea muy eficiente.

```
#include <iostream>
#include <string>
#include <cassert>
#include <array>
using namespace std ;

// -- Constantes -------
const int MAX_PERSONAS = 50 ;
// -- Tipos ------------
struct Direccion {
    unsigned num ;
    string calle ;
    string piso ;
    string cp ;
    string ciudad ;
} ;
struct Persona {
    string nombre ;
    string tel ;
    Direccion direccion ;
} ;
// -- Tipos ------------
typedef array<Persona, MAX_PERSONAS> Personas ;
struct Agenda {
    unsigned n_pers ;
    Personas pers ;
} ;
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
} ;
// -- Subalgoritmos ----
void inicializar (Agenda& ag)
{
    ag.n_pers = 0 ;
}
//---------------------------
void leer_direccion (Direccion& dir)
{
    cin >> dir.calle ;
    cin >> dir.num ;
    cin >> dir.piso ;
    cin >> dir.cp ;
    cin >> dir.ciudad ;
}
//---------------------------
void escribir_direccion (const Direccion& dir)
{
    cout << dir.calle << " " ;
    cout << dir.num << " " ;
    cout << dir.piso << " " ;
    cout << dir.cp << " " ;
    cout << dir.ciudad << " " ;
```

```
}
//--------------------------
void leer_persona (Persona& per)
{
    cin >> per.nombre ;
    cin >> per.tel ;
    leer_direccion(per.direccion) ;
}
//--------------------------
void escribir_persona (const Persona& per)
{
    cout << per.nombre << " " ;
    cout << per.tel << " " ;
    escribir_direccion(per.direccion) ;
    cout << endl ;
}
//--------------------------
// Busca una Persona en la Agenda Ordenada
// Devuelve su posición si se encuentra, o bien >= ag.n_pers en otro caso
unsigned buscar_persona (const string& nombre, const Agenda& ag)
{
    unsigned i = 0 ;
    unsigned f = ag.n_pers ;
    unsigned res = ag.n_pers ;
    while (i < f) {
        unsigned m = (i + f) / 2 ;
        int cmp = nombre.compare(ag.pers[m].nombre) ;
        if (cmp == 0) {
            res = m ;
            i = m ;
            f = m ;
        } else if (cmp < 0) {
            f = m ;
        } else {
            i = m + 1 ;
        }
    }
    return res ;
}
```

La implementación de `buscar_persona` pretende ser eficiente. Por ese motivo, evitamos repetir innecesariamente la comparación de la cadena a buscar y la cadena almacenada en la posición estudiada en cada paso. En su lugar, hacemos una única comparación y almacenamos su resultado en la variable `cmp`[1]. Posteriormente, en cada paso decidimos qué hacer en función del contenido de `cmp`.

Nótese que en este ejemplo hemos utilizado un algoritmo de búsqueda binaria diferente del presentado al principio del capítulo, consiguiendo así limitar el número de comparaciones de cadenas a realizar.

```
//--------------------------
unsigned buscar_posicion (const string& nombre, const Agenda& ag)
{
    unsigned i = 0 ;
    while ((i < ag.n_pers) && (nombre > ag.pers[i].nombre)) {
        ++i ;
    }
```

[1] Usamos la función `compare` de la biblioteca `string`, que devuelve 0 si las cadenas comparadas son iguales, un valor positivo si el argumento es lexicográficamente menor y un valor negativo en caso contrario.

```
    return i ;
}
//--------------------------
void anyadir_ord (Agenda& ag, unsigned pos, const Persona& per)
{
    for (unsigned i = ag.n_pers ; i > pos ; --i) {
        ag.pers[i] = ag.pers[i - 1] ;
    }
    ag.pers[pos] = per ;
    ++ag.n_pers ;
}
//--------------------------
void eliminar_ord (Agenda& ag, unsigned pos)
{
    --ag.n_pers ;
    for (unsigned i = pos ; i < ag.n_pers ; ++i) {
        ag.pers[i] = ag.pers[i + 1] ;
    }
}
//--------------------------
void anyadir_persona (const Persona& per, Agenda& ag, Cod_Error& ok)
{
    unsigned pos = buscar_posicion(per.nombre, ag) ;
    if ((pos < ag.n_pers) && (per.nombre == ag.pers[pos].nombre)) {
        ok = YA_EXISTE ;
    } else if (ag.n_pers >= ag.pers.size()) {
        ok = AG_LLENA ;
    } else {
        ok = OK ;
        anyadir_ord(ag, pos, per) ;
    }
}
//--------------------------
void borrar_persona (const string& nombre, Agenda& ag, Cod_Error& ok)
{
    unsigned i = buscar_persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        eliminar_ord(ag, i) ;
    }
}
//--------------------------
void modificar_persona (const string& nombre, const Persona& nuevo, Agenda& ag,
                        Cod_Error& ok)
{
    unsigned i = buscar_persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        eliminar_ord(ag, i) ;
        anyadir_persona(nuevo, ag, ok) ;
    }
}
//--------------------------
void imprimir_persona (const string& nombre, const Agenda& ag, Cod_Error& ok)
{
```

```
    unsigned i = buscar_persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        escribir_persona(ag.pers[i]) ;
    }
}
//--------------------------
void imprimir_agenda (const Agenda& ag, Cod_Error& ok)
{
    for (int i = 0 ; i < ag.n_pers ; ++i) {
        escribir_persona(ag.pers[i]) ;
    }
    ok = OK ;
}
//--------------------------
char menu ()
{
    char opcion ;
    cout << endl ;
    cout << "a. - Añadir Persona" << endl ;
    cout << "b. - Buscar Persona" << endl ;
    cout << "c. - Borrar Persona" << endl ;
    cout << "d. - Modificar Persona" << endl ;
    cout << "e. - Imprimir Agenda" << endl ;
    cout << "x. - Salir" << endl ;
    do {
        cout << "Introduzca Opción: " ;
        cin >> opcion ;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x'))) ;
    return opcion ;
}
//--------------------------
void escribir_cod_error (Cod_Error cod)
{
    switch (cod) {
    case OK:
        cout << "Operación correcta" << endl ;
        break ;
    case AG_LLENA:
        cout << "Agenda llena" << endl ;
        break ;
    case NO_ENCONTRADO:
        cout << "La persona no se encuentra en la agenda" << endl ;
        break ;
    case YA_EXISTE:
        cout << "La persona ya se encuentra en la agenda" << endl ;
        break ;
    }
}
// -- Principal --------
int main ()
{
    Agenda ag ;
    char opcion ;
    Persona per ;
    string nombre ;
    Cod_Error ok ;
```

```
    inicializar(ag) ;
    do {
        opcion = menu() ;
        switch (opcion) {
        case 'a':
            cout << "Introduzca los datos de la Persona"<<endl ;
            cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
            leer_persona(per) ;
            anyadir_persona(per, ag, ok) ;
            escribir_cod_error(ok) ;
            break ;
        case 'b':
            cout << "Introduzca Nombre" << endl ;
            cin >> nombre ;
            imprimir_persona(nombre, ag, ok) ;
            escribir_cod_error(ok) ;
            break ;
        case 'c':
            cout << "Introduzca Nombre" << endl ;
            cin >> nombre ;
            borrar_persona(nombre, ag, ok) ;
            escribir_cod_error(ok) ;
            break ;
        case 'd':
            cout << "Introduzca Nombre" << endl ;
            cin >> nombre ;
            cout << "Nuevos datos de la Persona" << endl ;
            cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
            leer_persona(per) ;
            modificar_persona(nombre, per, ag, ok) ;
            escribir_cod_error(ok) ;
            break ;
        case 'e':
            imprimir_agenda(ag, ok) ;
            escribir_cod_error(ok) ;
            break ;
        }
    } while (opcion != 'x' ) ;
}
```

Capítulo 8

Algunas Bibliotecas Útiles

En este capítulo se muestra superficialmente algunas funciones básicas de la biblioteca estándar.

cmath

La biblioteca `<cmath>` proporciona principalmente algunas funciones matemáticas útiles:

```
#include <cmath>
using namespace std ;
```

`double sin(double r) ;`	seno, $\sin r$ (en radianes)
`double cos(double r) ;`	coseno, $\cos r$ (en radianes)
`double tan(double r) ;`	tangente, $\tan r$ (en radianes)
`double asin(double x) ;`	arco seno, $\arcsin x, x \in [-1, 1]$
`double acos(double x) ;`	arco coseno, $\arccos x, x \in [-1, 1]$
`double atan(double x) ;`	arco tangente, $\arctan x$
`double atan2(double y, double x) ;`	arco tangente, $\arctan y/x$
`double sinh(double r) ;`	seno hiperbólico, $\sinh r$
`double cosh(double r) ;`	coseno hiperbólico, $\cosh r$
`double tanh(double r) ;`	tangente hiperbólica, $\tanh r$
`double sqrt(double x) ;`	$\sqrt{x}, x \geq 0$
`double pow(double x, double y) ;`	x^y
`double exp(double x) ;`	e^x
`double log(double x) ;`	logaritmo neperiano, $\ln x, x > 0$
`double log10(double x) ;`	logaritmo decimal, $\log x, x > 0$
`double ceil(double x) ;`	menor entero $\geq x$, $\lceil x \rceil$
`double floor(double x) ;`	mayor entero $\leq x$, $\lfloor x \rfloor$
`double fabs(double x) ;`	valor absoluto de x, $|x|$
`double ldexp(double x, int n) ;`	$x2^n$
`double frexp(double x, int* exp) ;`	inversa de ldexp
`double modf(double x, double* ip) ;`	parte entera y fraccionaria
`double fmod(double x, double y) ;`	resto de x/y

cctype

La biblioteca `<cctype>` proporciona principalmente características sobre los valores de tipo `char`:

```
#include <cctype>
using namespace std ;
```

`bool isalnum(char ch) ;`	`(isalpha(ch) || isdigit(ch))`
`bool isalpha(char ch) ;`	`(isupper(ch) || islower(ch))`
`bool iscntrl(char ch) ;`	caracteres de control
`bool isdigit(char ch) ;`	dígito decimal
`bool isgraph(char ch) ;`	caracteres imprimibles excepto espacio
`bool islower(char ch) ;`	letra minúscula
`bool isprint(char ch) ;`	caracteres imprimibles incluyendo espacio
`bool ispunct(char ch) ;`	carac. impr. excepto espacio, letra o dígito
`bool isspace(char ch) ;`	espacio, `'\r'`, `'\n'`, `'\t'`, `'\v'`, `'\f'`
`bool isupper(char ch) ;`	letra mayúscula
`bool isxdigit(char ch) ;`	dígito hexadecimal
`char tolower(char ch) ;`	retorna la letra minúscula correspondiente a `ch`
`char toupper(char ch) ;`	retorna la letra mayúscula correspondiente a `ch`

ctime

La biblioteca `<ctime>` proporciona principalmente algunas funciones generales relacionadas con el tiempo:

```
#include <ctime>
using namespace std ;
```

`clock_t clock() ;`	retorna el tiempo de CPU utilizado (`CLOCKS_PER_SEC`)
`time_t time(0) ;`	retorna el tiempo de calendario (en segundos)

```
#include <iostream>
#include <ctime>
using namespace std ;
// -------------------------------------
int main()
{
    time_t t1 = time(0) ;
    clock_t c1 = clock() ;
    // ... procesamiento ...
    clock_t c2 = clock() ;
    time_t t2 = time(0) ;
    cout << "Tiempo de CPU: " << double(c2 - c1)/double(CLOCKS_PER_SEC) << " seg" << endl ;
    cout << "Tiempo total: " << (t2 - t1) << " seg" << endl ;
}
// -------------------------------------
```

cstdlib

La biblioteca `<cstdlib>` proporciona principalmente algunas funciones generales útiles:

```
#include <cstdlib>
using namespace std ;
```

`int abs(int n) ;`	retorna el valor absoluto del número `int n`
`long labs(long n) ;`	retorna el valor absoluto del número `long n`
`int system(const char orden[]) ;`	orden a ejecutar por el sistema operativo
`void exit(int estado) ;`	termina la ejecución del programa actual (`EXIT_SUCCESS`, `EXIT_FAILURE`)
`void abort() ;`	aborta la ejecución del programa actual
`void srand(unsigned semilla) ;`	inicializa el generador de números aleatorios
`int rand() ;`	retorna un aleatorio entre `0` y `RAND_MAX` (ambos inclusive)

```
#include <cstdlib>
#include <ctime>
using namespace std ;
```

```
// -------------------------------------
// inicializa el generador de números aleatorios
inline void ini_aleatorio()
{
    srand(time(0)) ;
}
// -------------------------------------
// Devuelve un número aleatorio entre 0 y max (exclusive)
inline int aleatorio(int max)
{
    return int(max*double(rand())/(RAND_MAX+1.0)) ;
}
// -------------------------------------
// Devuelve un número aleatorio entre min y max (ambos inclusive)
inline int aleatorio(int min, int max)
{
    return min + aleatorio(max-min+1) ;
}
// -------------------------------------
```

Parte II

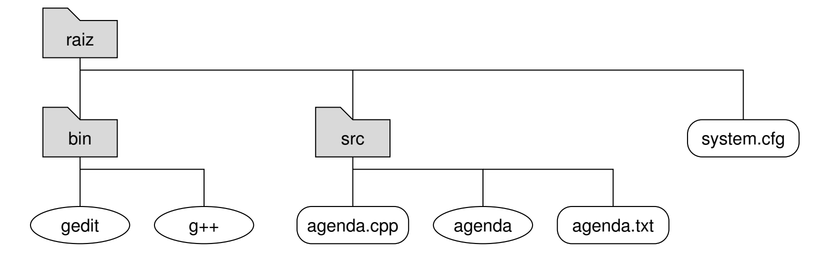
Programación Intermedia

Capítulo 9

Almacenamiento en Memoria Secundaria: Ficheros

Un programa suele trabajar con datos almacenados en la *memoria principal* (RAM). Ésta se caracteriza por proporcionar un acceso (para lectura y escritura) rápido a la información almacenada. Sin embargo, este tipo de memoria es *volátil*, en el sentido de que los datos almacenados en ella desaparecen cuando termina la ejecución del programa o se apaga el ordenador. Por este motivo, para almacenar información de manera *permanente* se utilizan dispositivos de almacenamiento de *memoria secundaria*, tales como dispositivos magnéticos (discos duros, cintas), discos ópticos (CDROM, DVD), memorias permanentes de estado sólido (memorias flash USB), etc.

Los dispositivos de memoria secundaria suelen disponer de gran capacidad de almacenamiento, por lo que es necesario alguna organización que permita gestionar y acceder a la información allí almacenada. A esta organización se la denomina el *sistema de ficheros*, y suele estar organizado jerárquicamente en directorios (a veces denominados también carpetas) y ficheros (a veces denominados también archivos). Los directorios permiten organizar jerárquicamente y acceder a los ficheros, y estos últimos almacenan de forma permanente la información, que puede ser tanto programas (software) como datos que serán utilizados por los programas.



Un determinado fichero se puede especificar indicando el **camino** (o ruta) a seguir para llegar hasta él dentro del sistema de ficheros. Existen dos mecanismos para especificar este camino: el absoluto y el relativo.

- En el camino absoluto, se especifica la secuencia de directorios por los que se debe pasar para llegar desde la raíz del sistema de ficheros (indicada por una barra `/` inicial) hasta el fichero determinado. Por ejemplo, para el fichero `agenda.txt`, podemos seguir el siguiente camino absoluto: `/src/agenda.txt`
- En el camino relativo, se parte desde una determinada posición en el sistema de ficheros, usualmente el directorio de trabajo, y se especifica la secuencia de directorios por los que se debe pasar para llegar desde la posición actual en el sistema de ficheros hasta el fichero determinado (nótese que el direccionamiento relativo no comienza por la barra `/` inicial), considerando además que el símbolo `..` representa al directorio padre del actual. Por ejemplo, si nos encontramos en el directorio de trabajo `/bin/`, podemos especificar el camino

hasta el fichero `agenda.txt` de forma relativa de la siguiente forma: `../src/agenda.txt`. Sin embargo, si nos encontramos en el directorio de trabajo `/src/`, entonces simplemente especificaremos el nombre del fichero `agenda.txt`, ya que se encuentra en el mismo directorio que el directorio de trabajo.

Tipos de Ficheros

Los ficheros se pueden clasificar atendiendo a diferentes criterios. En nuestro caso, nos centraremos en su clasificación en función de la codificación o formato en el que almacenan la información. Así, podemos distinguir entre *ficheros de texto* y *ficheros binarios*.

En los ficheros de texto la información se almacena como una secuencia de caracteres y cada carácter se almacena utilizando una codificación estándar (usualmente basada en la codificación ASCII, UTF-8, etc). Al tratarse de un formato estandarizado, otros programas diferentes de aquel que creó el fichero podrán entender y procesar su contenido. Por ejemplo, un programa podría generar un fichero de texto con los datos de las personas de una agenda y posteriormente dicho fichero podría ser entendido y procesado por otros programas. Por ejemplo, podría ser visualizado y editado mediante programas de edición de textos de propósito general, tales como `gedit`, `kate`, `gvim`, `emacs`, etc. en Linux, `textedit` en MacOS-X y `notepad` en Windows, entre otros.

En los ficheros binarios, la información se almacena con el mismo formato y codificación utilizada para su almacenamiento en memoria principal. Están concebidos para ser procesados automáticamente por programas que conocen su formato interno. Un programa no podrá procesar la información que contiene si no dispone de documentación adecuada que describa su formato interno[1]. El procesamiento de ficheros binarios es más eficiente que el de ficheros de texto porque la información está representada directamente en *código binario* (exactamente tal y como se encuentra internamente en la memoria principal). De esta forma se evita la pérdida de tiempo que ocasionaría su conversión a un formato estándar (ASCII, UTF-8, etc), como ocurre en los ficheros de texto.

En el caso del software, los programas en código fuente codificados en un lenguaje de programación suelen ser almacenados como ficheros de texto. Sin embargo, el resultado de compilar estos programas fuente a programas ejecutables se almacenan en ficheros binarios (ejecutables por el Sistema Operativo). Así mismo, los ficheros que contienen imágenes, vídeo y música suelen estar, en su mayoría, almacenados en formato binario.

Consideremos un ejemplo concreto y supongamos que disponemos de un fichero de texto denominado `fechas.txt`, que podría estar almacenado en una determinada posición en la jerarquía del sistema de ficheros (`/home/alumno/documentos/fechas.txt`) y contener información sobre las fechas de nacimiento de determinadas personas según el siguiente formato, donde cada línea se encuentra terminada por un carácter terminador de fin de línea:[2]

```
Juan López 12 3 1992
Lola Martínez 23 7 1987
Pepe Jiménez 17 8 1996
```

Aunque los datos almacenados en memoria se encuentran en formato binario, son convertidos a su representación textual antes de ser escritos en el fichero. Similarmente, cuando leemos del fichero de texto para almacenar la información en memoria se produce una conversión de formato de texto a fomato binario. Por ejemplo, si el número `12` se almacena en una variable de tipo `unsigned` su representación interna utilizaría un número binario de 32 bits (00000000000000000000000000001100). Sin embargo, su representación textual en el fichero de texto se compone de una secuencia de dos caracteres (`'1' '2'`). Como se puede observar, el valor se almacena internamente en memoria en formato binario utilizando 4 bytes, mientras que en el fichero de texto se almacenan los dos caracteres con el valor que corresponda según el código ASCII (2 bytes).

[1]Si un fichero binario es procesado por un programa que se ejecuta en un ordenador que utiliza una representación interna distinta de la utilizada en el ordenador en que se creó podemos tener problemas de compatibilidad.

[2]El carácter terminador de fin de línea *no es visible*, aunque se aprecian sus efectos al mostrarse los siguientes caracteres en la siguiente línea.

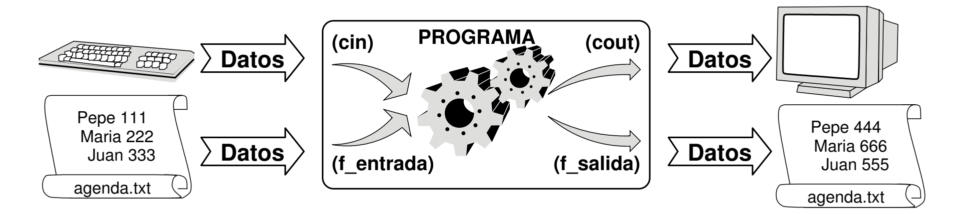
9.1. Flujos de Entrada y Salida Asociados a Ficheros

En el capítulo 3 se explicó que un programa codificado en C++ realiza la entrada y salida de información a través de flujos (*stream* en inglés) de entrada y salida respectivamente. Se estudió cómo realizar la entrada y salida de datos a través de los flujos estándares de entrada y salida (`cin` y `cout`, respectivamente), usualmente conectados con el teclado y la pantalla de la consola. Todo lo explicado anteriormente respecto a la entrada y salida básica con los flujos estándares (capítulo 3), o la entrada y salida de cadenas de caracteres (capítulo 6.2.1)también es aplicable a los flujos de entrada y salida vinculados a ficheros que veremos en este capítulo.

Un flujo de entrada de datos en modo texto actúa como una *fuente* que proporciona una *secuencia de caracteres* (usualmente a través de un buffer de almacenamiento intermedio) desde el que se extraen los caracteres que representan a los datos de entrada, que posteriormente serán convertidos a la representación interna adecuada.

```
Juan López 12 3 1992 ↩ Lola Martínez 23 7 1987 ↩ Pepe Jiménez 17 8 1996 ↩
```

Por el contrario, un flujo de salida de datos en modo texto actúa como un *sumidero* que recibe una *secuencia de caracteres* (usualmente a través de un buffer de almacenamiento intermedio) al que se envían los caracteres que representan a los datos de salida, que previamente han sido convertidos al formato de texto adecuado.



En el caso de entrada y salida a ficheros, el lenguaje de programación C++ posee mecanismos para asociar y *vincular* estos flujos con ficheros almacenados en memoria secundaria en el sistema de ficheros. Así, toda la entrada y salida de información se realiza a través de estos flujos vinculados a ficheros, denominados *manejadores de ficheros*. De este modo, una vez que un programa vincula un fichero con un determinado flujo de entrada o salida, las operaciones de lectura o escritura funcionan como ya hemos estudiado en los flujos estándar `cin` y `cout`.

Cuando un programa quiere realizar una entrada o salida de datos con un determinado fichero, debe realizar las siguientes acciones:

1. *Incluir* la biblioteca `<fstream>`, que contiene los elementos necesarios para procesar el fichero.
2. Usar el espacio de nombres `std`.
3. *Declarar* las variables que actuarán como manejadores de ficheros.
4. *Abrir* el flujo de datos, vinculando la variable correspondiente con el fichero especificado. Esta operación establece un vínculo entre la variable (manejador de fichero) definida en nuestro programa y el fichero gestionado por el sistema operativo. De esta forma, toda transferencia de información entre el programa y un fichero se realizará a través de la variable manejador que ha sido vinculada con dicho fichero.
5. *Comprobar* que la apertura del fichero del paso previo se realizó correctamente. Si la vinculación con el fichero especificado no pudo realizarse por algún motivo (por ejemplo, si queremos hacer entrada de datos de un fichero que no existe, o si es imposible crear un fichero en el que escribir datos), entonces la operación de apertura fallaría.
6. *Realizar* la transferencia de información (de entrada o de salida) con el fichero a través de la variable de flujo vinculada al mismo. Dicha transferencia de información se puede realizar utilizado los mecanismos vistos en los capítulos anteriores (3, 6.2.1). En el caso de salida, los datos deberán escribirse siguiendo un formato adecuado que permita su posterior lectura. Por ejemplo, escribiendo separadores adecuados entre los diferentes valores almacenados.

   Normalmente, tanto la entrada como la salida de datos se realizan mediante un proceso *iterativo*. En el caso de entrada dicho proceso suele requerir la lectura de todo el contenido del fichero.

7. *Comprobar* que el procesamiento del fichero del paso previo se realizó correctamente. En el caso de procesamiento para entrada ello suele consistir en comprobar si el estado de la variable manejador indica que se ha alcanzado el final del fichero. En el procesamiento para salida suele consistir en comprobar si el estado de la variable manejador indica que se ha producido un error de escritura.
8. Finalmente, *cerrar* el flujo para liberar la variable manejador de su vinculación con el fichero. En caso de no cerrar el flujo, éste será cerrado automáticamente cuando termine el ámbito de vida de la variable manejador del fichero.

Nota: es importante tener en cuenta que cuando un flujo pasa al estado erróneo (`fail()`), entonces cualquier operación de entrada o salida que se realice sobre él también fallará.

Las variables de tipo flujo pueden ser usadas como parámetros de subprogramas. En este caso hay que tener en cuenta que es necesario que, tanto si el fichero se quiere pasar como un parámetro de entrada, salida o entrada/salida, dicho paso de parámetro debe hacerse por referencia (**no** constante).

9.2. Entrada de Datos desde Ficheros de Texto

A continuación, presentaremos con más detalle y utilizando ejemplos concretos los pasos que se deben seguir para realizar la entrada de datos desde ficheros de texto.

1. Debemos asegurarnos que hemos incluido biblioteca `<fstream>` y que utilizamos el espacio de nombres `std`.

```
#include <fstream>
using namespace std;
```

2. Necesitamos definir una variable que actúe como manejador del fichero del que queremos leer. Esta variable debe ser de tipo `ifstream` (*input file stream*), disponible una vez que hemos importado la biblioteca `fstream`

```
ifstream f_ent;
```

3. La variable `f_ent` ha sido declarada para ser asociada a un determinado flujo, pero aún no hemos procedido a realizar una vinculación con un fichero concreto. Al proceso de vincular una variable manejador de fichero con un determinado fichero se le conoce como *abrir* el fichero.

```
f_ent.open("fechas.txt");
```

En este ejemplo hemos utilizando una constante literal de tipo cadena de caracteres para asociar el manejador `f_ent` con el fichero `fechas.txt`. En numerosas ocasiones el nombre del fichero a abrir no es siempre el mismo, podemos tener programas en los que, por ejemplo, el nombre del fichero a abrir sea uno concreto introducido por teclado. Para ello, podemos declarar la variable de tipo `string` correspondiente y utilizarla en el proceso de apertura. En tal caso debemos utilizar la función `c_str()` para adaptar la variable al tipo correcto antes de efectuar la llamada a `open`, como se muestra a continuación.

```
string nom_fich;

cout << "Nombre del fichero a abrir: ";
cin >> nom_fich;
f_ent.open(nom_fich.c_str());
```

4. Comprobar que la apertura del fichero se realizó correctamente y evitar el procesamiento del fichero en caso de fallo en la apertura.

```
if (f_ent.fail()) { ... }
```

5. Realizar la entrada de datos con los operadores y subprogramas correspondientes, así como procesar la información leída. Por ejemplo, para un fichero con el formato mostrado en la sección 9, podríamos leer dos cadenas con el nombre y apellidos de la persona (hasta el primer separador) y tres números con el día, mes y año de nacimiento de la persona[3].

[3]Suponemos que cada uno de los datos leídos se encuentran seguidos de un separador en el fichero.

```
f_ent >> nombre >> apellidos >> dia >> mes >> anyo;
```

o bien podríamos estar interesados en leer una línea completa en una variable de tipo `string`.

```
string linea;
getline(f_ent, linea);
```

En el capítulo 6.2.1 estudiamos que la lectura de datos puede requerir limpiar el buffer de entrada en determinada situaciones. Ahora, aunque el flujo desde el que efectuar la entrada es un fichero (en lugar de `cin`) el comportamiento es exactamente el mismo que estudiamos en dicho capítulo. Por ese motivo, si utilizamos `getline` para leer una cadena del fichero, es posible que necesitemos omitir (*limpiar*) determinados caracteres del buffer de entrada del fichero para asegurarnos de que la lectura se realiza correctamente. En tal caso, utilizaríamos el manipulador `ws` o bien la función `ignore` (sobre la variable manejador del fichero) para limpiar el buffer de entrada, al igual que hicimos en dicho capítulo.

```
f_ent >> edad ;
f_ent.ignore(1000, '\n') ;
//---------------------------
getline(f_ent, linea) ;
```

```
f_ent >> edad ;
//---------------------------
f_ent >> ws ;
getline(f_ent, linea) ;
```

Al igual que podemos usae la función `get` para leer un carácter de teclado, podemos utilizar dicha función para leer un carácter de un fichero. En tal caso deberemos indicar el manejador del fichero del que queremos realizar la lectura.

```
char car;
f_ent.get(car);
```

Usualmente la lectura de datos de un fichero se realiza dentro de un proceso iterativo que acaba cuando el fichero no contiene datos que procesar. Necesitamos algún modo de detectar que se ha alcanzado el fin de un fichero durante un proceso de lectura. Para ello, comprobaremos si se ha producido un fallo al intentar efectuar una operación de lectura. Por lo general, este proceso iterativo suele responder al siguiente esquema general:

- Lectura de datos
- Si la lectura no ha sido correcta, entonces terminar el proceso iterativo.
- En otro caso, realizamos el procesamiento de los datos leídos, y continuamos el proceso iterativo, leyendo nuevos datos

```
...
f_ent >> datos;
while (! f_ent.fail() ... ) {
    procesar(datos, ...);
    f_ent >> datos;
}
```

6. Comprobar que el procesamiento del fichero se realizó correctamente, es decir, que el fichero se leyó completamente hasta el final de mismo (`eof` representa *end-of-file*). Si el fichero no acabó correctamente es porque se produjo un error de lectura durante su procesamiento. En tal caso, posiblemente nos interesará tratar adecuadamente dicha situación de error.

```
if (!f_ent.fail() || f_ent.eof()) { /* OK */ }
```

7. Finalmente, cerrar el flujo liberando la variable de su vinculación.

```
f_ent.close();
```

Acabaremos esta sección mostrando un ejemplo completo en el que utilizamos todos los elementos introducidos anteriormente. Queremos leer números[4] de un fichero y mostrarlos en pantalla. En numerosas ocasiones el esquema del programa mostrado a continuación puede ser utilizado en otros programas. Para ello bastaría con modificar el subprograma `leer`, adaptando la lectura al caso concreto que deseemos tratar, y el subprograma `procesar` para el procesamiento que deseemos realizar con los datos leídos.

[4] Se supone que los números están separados por espacios o separadores adecuados.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
enum Codigo {
    OK, ERROR_APERTURA, ERROR_FORMATO
};
void procesar(int num)
{
    cout << num << endl;
}
void leer(ifstream& fich, int& num)
{
    fich >> num;
}
void leer_fich(const string& nombre_fichero, Codigo& ok)
{
    ifstream f_ent;
    f_ent.open(nombre_fichero.c_str());
    if (f_ent.fail()) {
        ok = ERROR_APERTURA;
    } else {
        int numero;
        leer(f_ent, numero);
        while (! f_ent.fail()) {
            procesar(numero);
            leer(f_ent, numero);
        }
        if (!f_ent.fail() || f_ent.eof()) {
            ok = OK;
        } else {
            ok = ERROR_FORMATO;
        }
        f_ent.close();
    }
}
void codigo_error(Codigo ok)
{
    switch (ok) {
    case OK:
        cout << "Fichero procesado correctamente" << endl;
        break;
    case ERROR_APERTURA:
        cout << "Error en la apertura del fichero" << endl;
        break;
    case ERROR_FORMATO:
        cout << "Error de formato en la lectura del fichero" << endl;
        break;
    }
}
int main()
{
    Codigo ok;
    string nombre_fichero;
    cout << "Introduzca el nombre del fichero: ";
    cin >> nombre_fichero;
    leer_fich(nombre_fichero, ok);
    codigo_error(ok);
}
```

9.3. Salida de Datos a Ficheros de Texto

Para escribir datos en ficheros de texto utilizamos los mismos elementos que para realizar la lectura, pero considerando las siguientes diferencias:

1. Debemos asegurarnos que hemos incluido biblioteca `<fstream>` y que utilizamos el espacio de nombres `std`.

   ```
   #include <fstream>
   using namespace std;
   ```

2. Necesitamos definir una variable que actúe como manejador del fichero al que queremos escribir. Esta variable debe ser de tipo `ofstream` (*output file stream*), disponible una vez que hemos importado la biblioteca `fstream`

   ```
   ofstream f_sal;
   ```

3. La variable `f_sal` ha sido declarada para ser asociada a un determinado flujo, pero aún no hemos procedido a realizar una vinculación con un fichero concreto. Al proceso de vincular una variable manejador de fichero con un determinado fichero se le conoce como *abrir* el fichero.

   ```
   f_sal.open("fechas.txt");
   ```

   En este ejemplo hemos utilizando una constante literal de tipo cadena de caracteres para asociar el manejador `f_sal` con el fichero `fechas.txt`. En numerosas ocasiones el nombre del fichero a abrir no es siempre el mismo, podemos tener programas en los que, por ejemplo, el nombre del fichero a abrir sea uno concreto introducido por teclado. Para ello, podemos declarar la variable de tipo `string` correspondiente y utilizarla en el proceso de apertura. En tal caso debemos utilizar la función `c_str()` para adaptar la variable al tipo correcto antes de efectuar la llamada a `open`, como se muestra a continuación.

   ```
   string nom_fich;

   cout << "Nombre del fichero a abrir: ";
   cin >> nom_fich;
   f_sal.open(nom_fich.c_str());
   ```

4. Comprobar que la apertura del fichero se realizó correctamente y evitar el procesamiento del fichero en caso de fallo en la apertura.

   ```
   if (f_sal.fail()) { ... }
   ```

5. Si escribimos datos en un fichero es con la intención de proceder en un futuro a su lectura. Así pues, debemos escribir los datos con los separadores adecuados entre ellos, permitiendo así que futuras lecturas del mismo funcionen correctamente. Por ejemplo, si queremos escribir datos de una persona en un fichero como el usado en la sección anterior, será necesario escribir los separadores entre cada dos valores. Por ejemplo, a continuación utilizamos *espacios en blanco* entre cada dos datos de una persona y un salto de línea para separar una persona de la siguiente.

   ```
   f_sal << nombre << " " << apellidos << " "
         << dia << " " << mes << " " << anyo << endl;
   ```

   Usualmente la escritura de datos se realiza mediante un proceso iterativo que finaliza cuando se escriben en el fichero todos los datos apropiados y mientras el estado del flujo sea correcto.

   ```
   while ( ... !f_sal.fail() ) {
       ...
   }
   ```

6. Al acabar el bucle es conveniente comprobar si se ha producido algún error durante la escritura en el fichero (el flujo está en un estado incorrecto) y en su caso, procesarlo adecuadamente.

   ```
   if (!f_sal.fail()) { /* OK */ }
   ```

7. Finalmente, cerrar el flujo liberando la variable de su vinculación.

```
f_sal.close();
```

Por ejemplo, a continuación se muestra un programa que lee números de teclado (hasta introducir un cero) y los escribe en un fichero de texto. Los números se escriben separados por un carácter de fin de línea (`endl`), lo que permite que el fichero aparezca con cada número en una línea diferente si es mostrado por un editor o que los números puedan ser leídos y procesados por un programa como el presentado en el ejemplo anterior.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
enum Codigo {
    OK, ERROR_APERTURA, ERROR_FORMATO
};
void escribir(ofstream& f_sal, int num)
{
    f_sal << num << endl ;
}
void escribir_fich(const string& nombre_fichero, Codigo& ok)
{
    ofstream f_sal;
    f_sal.open(nombre_fichero.c_str());
    if (f_sal.fail()) {
        ok = ERROR_APERTURA;
    } else {
        int numero;
        cin >> numero;
        while ((numero != 0) && ! cin.fail() && ! f_sal.fail()) {
            escribir(f_sal, numero);
            cin >> numero;
        }
        if (!f_sal.fail()) {
            ok = OK;
        } else {
            ok = ERROR_FORMATO;
        }
        f_sal.close();
    }
}
```

Hemos utilizado un bucle de escritura que acaba cuando se introduce un cero por teclado. Estamos acostumbrados a resolver problemas como éste, en los que trabajamos con secuencias de números que se leen de teclado hasta que se cumple una cierta condición. En este caso hemos incluido dos condiciones adicionales en la expresión que controla el fin del bucle (`! cin.fail() && ! f_sal.fail()`).

Con la primera condición (`! cin.fail()`) nos aseguramos de que el proceso iterativo se detiene si se produce algún error durante la lectura de teclado y el flujo `cin` entra en un estado incorrecto. Hasta ahora siempre hemos supuesto que el usuario introduce datos correctos, por lo que no hemos controlado posibles errores en la entrada. Sin embargo, esta suposición puede resultar *peligrosa* en un programa como el mostrado en este ejemplo porque, en caso de ocurrir alguún error de lectura (por ejemplo, si en lugar de introducir un número de tipo `int` el usuario introduce una cadena de caracteres), el flujo de entrada `cin` entraría en un estado de error, dando lugar a un bucle infinito que podría hacer que el fichero creciera sin control hasta ocupar todo el espacio disponible en el dispositivo que almacena el fichero. La inclusión de esta condición evita este riesgo.

Con la segunda condición (`! f_sal.fail()`) nos aseguramos de que el proceso iterativo se detiene si se produce algún error durante la escritura en el fichero, por ejemplo, si el dispositivo no dispone de memoria suficiente.

```
void codigo_error(Codigo ok)
```

```
{
    switch (ok) {
    case OK:
        cout << "Fichero guardado correctamente" << endl;
        break;
    case ERROR_APERTURA:
        cout << "Error en la apertura del fichero" << endl;
        break;
    case ERROR_FORMATO:
        cout << "Error de formato al escribir al fichero" << endl;
        break;
    }
}
int main()
{
    Codigo ok;
    string nombre_fichero;
    cout << "Introduzca el nombre del fichero: ";
    cin >> nombre_fichero;
    escribir_fich(nombre_fichero, ok);
    codigo_error(ok);
}
```

9.4. Ejemplos

Ejemplo 1. Copia del contenido de un fichero en otro.

A continuación, mostramos un programa que lee carácter a carácter el contenido de un fichero de texto y crea un nuevo fichero con el mismo contenido. El programa se basa en el subprograma `copiar_fichero`, que recibe dos cadenas de caracteres con el nombre de los ficheros origen y destino, realiza la copia y devuelve el estado de error resultante de efectuar la operación.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

enum Codigo {
    OK, ERROR_APERTURA_ENT, ERROR_APERTURA_SAL, ERROR_FORMATO
};
void copiar_fichero(const string& salida, const string& entrada, Codigo& ok)
{
    ifstream f_ent;
    f_ent.open(entrada.c_str());
    if (f_ent.fail()) {
        ok = ERROR_APERTURA_ENT;
    } else {
        ofstream f_sal;
        f_sal.open(salida.c_str());
        if (f_sal.fail()) {
            ok = ERROR_APERTURA_SAL;
        } else {
            char ch;
            f_ent.get(ch);
            while (! f_ent.fail() && ! f_sal.fail()) {
                f_sal << ch;
                f_ent.get(ch);
            }
            if ((! f_ent.fail() || f_ent.eof()) && ! f_sal.fail()) {
```

```
                ok = OK;
            } else {
                ok = ERROR_FORMATO;
            }
            f_sal.close();
        }
        f_ent.close();
    }
}
```

En este ejemplo trabajamos con dos ficheros simultáneamente, uno para entrada y otro para salida, por lo que usamos dos manejadores diferentes. Al igual que en los ejemplos anteriores, antes de trabajar con cada fichero hay que asociar su nombre con el manejador correspondiente, posteriormente se efectúan las operaciones que corresponda con cada uno y, finalmente, se liberan las vinculaciones de los manejadores con los ficheros[5]. En este caso el procesamiento del fichero de entrada en cada paso consiste en leer un carácter (usamos `get` porque queremos leer también los separadores) y el procesamiento del fichero de salida consiste en escribir un único carácter.

```
void codigo_error(Codigo ok)
{
    switch (ok) {
    case OK:
        cout << "Fichero procesado correctamente" << endl;
        break;
    case ERROR_APERTURA_ENT:
        cout << "Error en la apertura del fichero de entrada" << endl;
        break;
    case ERROR_APERTURA_SAL:
        cout << "Error en la apertura del fichero de salida" << endl;
        break;
    case ERROR_FORMATO:
        cout << "Error de formato en la lectura del fichero" << endl;
        break;
    }
}
int main()
{
    Codigo ok;
    string entrada, salida;
    cout << "Introduzca el nombre del fichero de entrada: ";
    cin >> entrada;
    cout << "Introduzca el nombre del fichero de salida: ";
    cin >> salida;
    copiar_fichero(salida, entrada, ok);
    codigo_error(ok);
}
```

Ejemplo 2

Ejemplo de un programa que crea, guarda y carga una agenda personal.

```
//-------------------------------------------------------------------------
#include <iostream>
#include <fstream>
#include <string>
```

[5]Los manejadores de ficheros son en este caso variables locales, que se destruyen automáticamente al acabar el subprograma, liberando los recursos necesarios. Así pues, en este caso podríamos haber omitido la liberación explícita (`close`). Sin embargo, optamos por incluirla porque ello refleja claramente el proceso a seguir en cualquier manipulación de ficheros (`abrir, usar, cerrar`)

```
#include <array>
#include <cctype>
using namespace std ;
//------------------------------------------------------------------------
struct Fecha {
    unsigned dia ;
    unsigned mes ;
    unsigned anyo ;
} ;
struct Persona {
    string nombre ;
    string tfn ;
    Fecha  fnac ;
} ;
const int MAX = 100 ;
typedef array<Persona, MAX> APers ;
struct Agenda {
    int nelms ;
    APers elm ;
} ;
//------------------------------------------------------------------------
void inic_agenda(Agenda& ag)
{
    ag.nelms = 0 ;
}
void anyadir_persona(Agenda& ag, const Persona& p, bool& ok)
{
    if (ag.nelms < int(ag.elm.size())) {
        ag.elm[ag.nelms] = p ;
        ++ag.nelms ;
        ok = true ;
    } else {
        ok = false ;
    }
}
//------------------------------------------------------------------------
void leer_fecha(Fecha& f)
{
    cout << "Introduza fecha de nacimiento (dia mes año): " ;
    cin >> f.dia >> f.mes >> f.anyo ;
}
void leer_persona(Persona& p)
{
    cout << "Introduza nombre: " ;
    cin >> ws ;
    getline(cin, p.nombre) ;
    cout << "Introduza teléfono: " ;
    cin >> p.tfn ;
    leer_fecha(p.fnac) ;
}
void nueva_persona(Agenda& ag)
{
    bool ok ;
    Persona p ;
    leer_persona(p) ;
    if (! cin.fail()) {
        anyadir_persona(ag, p, ok) ;
        if (!ok) {
            cout << "Error al introducir la nueva persona" << endl ;
```

```
        }
    } else {
        cout << "Error al leer los datos de la nueva persona" << endl ;
        cin.clear() ;
        cin.ignore(1000, '\n') ;
    }
}
//-------------------------------------------------------------------------
void escribir_fecha(const Fecha& f)
{
    cout << f.dia << '/' << f.mes << '/' << f.anyo ;
}
void escribir_persona(const Persona& p)
{
    cout << "Nombre: " << p.nombre << endl ;
    cout << "Teléfono: " << p.tfn << endl ;
    cout << "Fecha nac: " ;
    escribir_fecha(p.fnac) ;
    cout << endl ;
}
void escribir_agenda(const Agenda& ag)
{
    for (int i = 0 ; i < ag.nelms ; ++i) {
        cout << "----------------------------------------" << endl ;
        escribir_persona(ag.elm[i]) ;
    }
    cout << "----------------------------------------" << endl ;
}
//-------------------------------------------------------------------------
// FORMATO DEL FICHERO DE ENTRADA:
//
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  ...
//-------------------------------------------------------------------------
void leer_fecha(ifstream& fich, Fecha& f)
{
    fich >> f.dia >> f.mes >> f.anyo ;
}
void leer_persona(ifstream& fich, Persona& p)
{
    fich >> ws ;
    getline(fich, p.nombre) ;
    fich >> p.tfn ;
    leer_fecha(fich, p.fnac) ;
}
//----------------------------------------------
// Otra posible implementación
// void leer_persona(ifstream& fich, Persona& p)
// {
//     getline(fich, p.nombre) ;
//     fich >> p.tfn ;
//     leer_fecha(fich, p.fnac) ;
//     fich.ignore(1000, '\n') ;
// }
//----------------------------------------------
void leer_agenda(const string& nombre_fich, Agenda& ag, bool& ok)
```

```
{
    ifstream fich ;
    Persona p ;

    fich.open(nombre_fich.c_str()) ;
    if (fich.fail()) {
        ok = false ;
    } else {
        ok = true ;
        inic_agenda(ag) ;
        leer_persona(fich, p) ;
        while (!fich.fail() && ok) {
            anyadir_persona(ag, p, ok) ;
            leer_persona(fich, p) ;
        }
        ok = ok && (!fich.fail() || fich.eof()) ;
        fich.close() ;
    }
}
void cargar_agenda(Agenda& ag)
{
    bool ok ;
    string nombre_fich ;
    cout << "Introduce el nombre del fichero: " ;
    cin >> nombre_fich ;
    leer_agenda(nombre_fich, ag, ok) ;
    if (!ok) {
        cout << "Error al cargar el fichero" << endl ;
    }
}
//-------------------------------------------------------------------------
// FORMATO DEL FICHERO DE SALIDA:
//
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  ...
//-------------------------------------------------------------------------
void escribir_fecha(ofstream& fich, const Fecha& f)
{
    fich << f.dia << ' ' << f.mes << ' ' << f.anyo ;
}
void escribir_persona(ofstream& fich, const Persona& p)
{
    fich << p.nombre << endl ;
    fich << p.tfn << ' ' ;
    escribir_fecha(fich, p.fnac) ;
    fich << endl ;
}
void escribir_agenda(const string& nombre_fich, const Agenda& ag, bool& ok)
{
    ofstream fich ;

    fich.open(nombre_fich.c_str()) ;
    if (fich.fail()) {
        ok = false ;
    } else {
        int i = 0 ;
```

```
        while ((i < ag.nelms) && (! fich.fail())) {
            escribir_persona(fich, ag.elm[i]) ;
            ++i ;
        }
        ok = ! fich.fail() ;
        fich.close() ;
    }
}
void guardar_agenda(const Agenda& ag)
{
    bool ok ;
    string nombre_fich ;
    cout << "Introduce el nombre del fichero: " ;
    cin >> nombre_fich ;
    escribir_agenda(nombre_fich, ag, ok) ;
    if (!ok) {
        cout << "Error al guardar el fichero" << endl ;
    }
}
//--------------------------------------------------------------------------
char menu()
{
    char op ;
    cout << endl ;
    cout << "C. Cargar Agenda" << endl ;
    cout << "M. Mostrar Agenda" << endl ;
    cout << "N. Nueva Persona" << endl ;
    cout << "G. Guardar Agenda" << endl ;
    cout << "X. Fin" << endl ;
    do {
        cout << endl << "    Opción: " ;
        cin >> op ;
        op = char(toupper(op)) ;
    } while (!((op == 'C')||(op == 'M')||(op == 'N')||(op == 'G')||(op == 'X'))) ;
    cout << endl ;
    return op ;
}
//--------------------------------------------------------------------------
int main()
{
    Agenda ag ;
    char op ;
    inic_agenda(ag) ;
    do {
        op = menu() ;
        switch (op) {
        case 'C':
            cargar_agenda(ag) ;
            break ;
        case 'M':
            escribir_agenda(ag) ;
            break ;
        case 'N':
            nueva_persona(ag) ;
            break ;
        case 'G':
            guardar_agenda(ag) ;
            break ;
        }
```

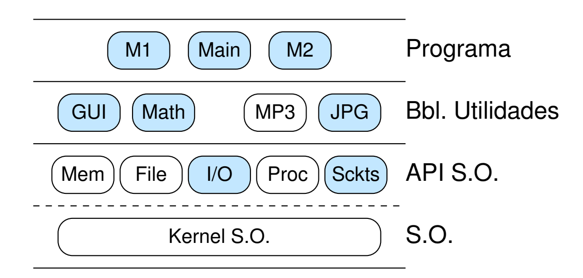
```
    } while (op != 'X') ;
}
//--------------------------------------------------------------------------
```

Capítulo 10

Módulos y Bibliotecas

Cuando se desarrollan programas de complejidad media/alta, el código fuente no suele encontrarse en un único fichero, sino distribuido entre varios módulos. En este contexto, un *módulo* es una entidad que agrupa diferentes elementos (subprogramas, tipos, constantes) interrelacionados entre sí. El programa completo se forma mediante la composición de diferentes módulos y, a su vez, un módulo puede incluir a otros si necesita hacer uso de los elementos definidos en éstos. La estructuración de un programa en módulos presenta numerosas ventajas:

- Permite aumentar la localidad y cohesión del código, aislándolo del exterior. Es decir, permite separar y aislar el código encargado de resolver un determinado problema.
- Facilita la *compilación separada*. Si se necesita modificar algún elemento interno a un módulo, no será necesario volver a compilar todo el programa sino únicamente la parte afectada por dicho cambio. Ello es especialmente importante en programas grandes, en los que el tiempo de compilación puede ser considerable.
- Facilita la *reutilización del código*. Es posible, tanto utilizar las bibliotecas del sistema como crear módulos con nuevas bibliotecas que puedan ser reutilizadas por múltiples programas. Esta distribución de bibliotecas se puede hacer en *código objeto*, por lo que no es necesario distribuir el código fuente de la misma.



En la figura se muestra un determinado programa que resuelve un determinado problema, cuya solución principal se ha dividido en varios módulos de programa (`Main`, `M1` y `M2`). Además, se hace uso de varios módulos de biblioteca que proporcionan utilidades gráficas, matemáticas y de tratamiento de imágenes; otros módulos de biblioteca son proporcionados por el sistema operativo y dan acceso a servicios de entrada/salida y comunicaciones por Internet.

10.1. Interfaz e Implementación del Módulo

En el lenguaje de programación C++, normalmente un módulo se compone de dos ficheros: uno donde aparece el código que resuelve un determinado problema o conjunto de problemas (la parte

privada), y otro que contiene las definiciones de tipos, constantes y prototipos de subprogramas que el módulo ofrece (la parte pública). Hablaremos de la *implementación* del módulo cuando nos refiramos al fichero que contiene su parte privada, y usaremos el témino *interfaz* del módulo para referirmos al fichero que contiene su parte pública[1].

Normalmente un programa completo se compone de varios módulos, cada uno con su fichero de encabezamiento (interfaz) y con su fichero de implementación, y de un módulo principal donde reside la función principal main. Los ficheros de implementación tendrán una extensión ".cpp" (también suelen utilizarse otras extensiones como ".cxx" y ".cc") y los ficheros de encabezamiento tendrán una extensión ".hpp" (también suelen utilizarse otras extensiones como ".hxx", ".hh" y ".h").

Cuando en un determinado módulo se desee hacer uso de las utilidades proporcionadas por otro módulo, éste deberá incluir el fichero de encabezamiento (interfaz) del módulo que se vaya a utilizar. Además, el fichero de implementación de un determinado módulo deberá incluir al fichero de encabezamiento de su propio módulo. Por ejemplo, si quisiéramos obtener un programa en el que se trabaje con números complejos, podríamos pensar en disponer de un módulo que se encargue de definir el tipo Complejo junto a una serie de operaciones. En el fichero de interfaz (*complejo.hpp*) se encontraría la definición del tipo y la declaración de los subprogramas y en el fichero de implementación (*complejo.cpp*) se encontraría la implementación de los subprogramas. El módulo principal debería importar el fichero de interfaz (*complejo.hpp*) para poder hacer uso del tipo Complejo y de los subprogramas. Además, el módulo de implementación también debería importar a su fichero de interfaz para poder conocer aquello que se desea implementar. El siguiente dibujo muestra el esquema básico de los ficheros usados en este ejemplo, donde se utiliza la directiva #include para incluir los ficheros de encabezamiento que corresponda:

```
main.cpp (Principal)
#include "complejo.hpp"
// utilización de complejo
int main()
{
   ···
}
```

```
complejo.hpp (Interfaz)
#ifndef complejo_hpp_
#define complejo_hpp_
// interfaz de complejo
// público
···
#endif
```

```
complejo.cpp (Implementación)
#include "complejo.hpp"
// implementación de complejo
// privado
···
···
···
```

La inclusión de ficheros de encabezamiento en nuestros módulos de programa no es algo nuevo para nosotros. Estamos acostumbrados a hacerlo cuando necesitamos efectuar entrada/salida con la consola (<iostream>), trabajar con cadenas de caracteres (<string>) o con con ficheros (<fstream>). En estos casos omitimos la extensión del fichero de encabezamiento y usamos <...> para delimitar el nombre del fichero a incluir. Lo hemos hecho de esta forma porque se trata de bibliotecas estándares, de uso muy frecuente e instaladas en una localización típica del sistema de archivos. Cuando el compilador detecta que el nombre del fichero a incluir está delimitado por <...>, sabe que se trata de una de tales bibliotecas y dónde localizar el fichero correspondiente. En nuestro ejemplo, el módulo para trabajar con números complejos no está instalado con el resto de bibliotecas estándares, sino que sus ficheros de encabezamiento e implementación se encuentran en el directorio de trabajo, por lo que usamos "..." para delimitar el fichero a incluir, indicando así al compilador dónde localizar los ficheros necesarios.

Guardas en un Fichero de Encabezamiento

Las definiciones en los ficheros de encabezamiento (interfaz) serán especificadas entre las *guardas* (directivas de compilación condicional) para evitar la inclusión duplicada de las definiciones allí contenidas, durante la compilación de un determinado módulo. El nombre de la guarda usualmente se deriva del nombre del fichero, como se indica en el siguiente ejemplo donde el módulo complejo tendrá los siguientes ficheros de encabezamiento y de implementación (en determinadas circunstancias, puede ser conveniente que al nombre de la guarda se le añada también el nombre del espacio de nombres que se explicará en la siguiente sección):

[1] A este fichero también se le denomina "fichero de encabezamiento" o "fichero de cabecera" (*header file* en inglés).

Fichero: `complejo.hpp` (Interfaz)	Fichero: `complejo.cpp` (Implementación)

`// Guarda para evitar inclusión duplicada`
`#ifndef complejo_hpp_`
`#define complejo_hpp_`
 	
`// Definiciones Públicas de:`
`//   * Constantes`
`//   * Tipos (Enum, Registros, Clases)`
`//   * Prototipos de Subprogramas`
 	
`#endif // Fin de guarda`
	`#include "complejo.hpp"`
	
	`// Implementaciones Privadas de:`
	`//   * Constantes Privadas`
	`//   * Tipos Privados`
	`//   * Subprogramas`
	`//   * Clases`

Directrices para el Diseño de Ficheros de Encabezamiento

Con objeto de organizar adecuadamente el diseño de los ficheros de encabezamiento de los módulos, tendremos en cuenta las siguientes directrices:

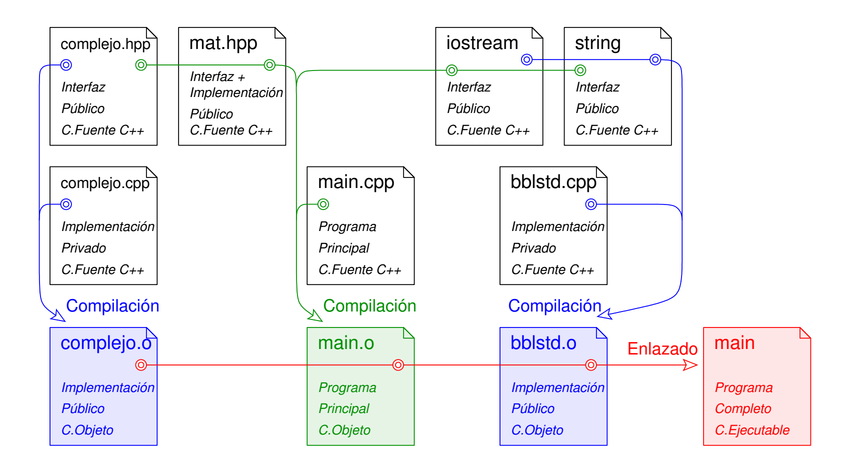
- Un fichero de encabezamiento sólo deberá contener definiciones de constantes, definiciones de tipos y prototipos de los subprogramas que exporta (parte pública) el módulo. No deberá contener definiciones de variables globales, ni la implementación de código (de subprogramas y métodos). En este último caso se contemplan algunas excepciones, tales como la definición de subprogramas simples "en línea" (véase 5.6) y la definición de subprogramas y clases genéricas. .

- El mecanismo de inclusión de ficheros de encabezamiento debe ser robusto ante posibles inclusiones duplicadas. Para ello, siempre se utilizará el mecanismo de guardas explicado anteriormente.

- Un fichero de encabezamiento debe incluir todos los ficheros de encabezamiento de otros módulos que necesite para su propia definición. De esta forma, no importará el orden de inclusión de los ficheros de encabezamiento, ya que cada fichero contiene todo lo necesario para su compilación.

10.2. Compilación Separada y Enlazado

El diseño de un programa mediante su organización en módulos permite compilar de forma separada los diferentes módulos que lo componen. Ello contribuye a que el proceso de compilación sea más flexible y rápido. Es frecuente que estemos trabajando con un programa en el que pretendemos introducir algún cambio o corregir algún error. En este caso, posiblemente habremos compilado el programa previamente y dispongamos del código objeto de la versión anterior. Si las modificaciones realizadas únicamente afectan a un determinado módulo, no es necesario volver a compilar el código fuente del resto de módulos. Bastará con compilar el código fuente del módulo afectado y enlazar el código objeto obtenido con el código objeto del resto de los módulos del programa.



Para compilar un módulo de forma independiente, únicamente hay que con compilar su fichero de implementación, por ejemplo `complejo.cpp`. El resultado es un fichero en *código objeto*, por ejemplo `complejo.o`. El compilador compila un código fuente que será el resultado de incluir en el fichero de implementación el contenido de los ficheros de encabezamiento, como se indica en la figura anterior. En cualquier caso, una vez que tenemos los ficheros objetos de los diferentes módulos, se procede a su enlazado final, obteniendo el fichero ejecutable.

Dependiendo del entorno en el que trabajemos, la tarea de compilación podrá ser realizada, bien directamente desde la línea de comando o con herramientas adecuadas en un entorno de desarrollo integrado. A continuación mostramos algunas de las opciones que se pueden realizar, suponiendo que trabajamos directamente desde la línea de comando y que utilizamos el compilador *GNU GCC*.

Podríamos generar los ficheros objetos de los dos módulos de nuestro programa (`complejo.o` y `main.o`) de la siguiente forma:

```
g++ -ansi -Wall -Werror -c complejo.cpp
g++ -ansi -Wall -Werror -c main.cpp
```

y enlazar los códigos objeto generados en el punto anterior para generar el fichero ejecutable `main`, con el siguiente comando:

```
g++ -ansi -Wall -Werror -o main main.o complejo.o
```

Hay otras posibilidades, también es posible realizar la compilación y enlazado en el mismo comando:

```
g++ -ansi -Wall -Werror -o main main.cpp complejo.cpp
```

o incluso mezclar compilación de código fuente y enlazado de código objeto:

```
g++ -ansi -Wall -Werror -o main main.cpp complejo.o
```

Hay que tener en cuenta que el compilador enlaza automáticamente el código generado con las bibliotecas estándares de C++ y, por lo tanto, no es necesario que éstas se especifiquen explícitamente. Sin embargo, en caso de ser necesario, también es posible especificar el enlazado con bibliotecas externas:

```
g++ -ansi -Wall -Werror -o main main.cpp complejo.cpp -ljpeg
```

Estas bibliotecas no son más que una agregación de módulos compilados a código objeto, y organizadas adecuadamente para que puedan ser reutilizados por muy diversos programas.

10.3. Espacios de Nombres

Cuando se trabaja con múltiples módulos y bibliotecas, es posible que se produzcan *colisiones* en los identificadores utilizados para nombrar a diferentes entidades. Es posible que un mismo

identificador sea utilizado en un módulo o biblioteca para nombrar a una determinada entidad y en otro módulo diferente se utilice para nombrar a otra entidad diferente. Por ejemplo, podríamos tener una biblioteca que defina en su interfaz una constante llamada `MAX`, con valor 100, y otra biblioteca diferente que utilice el mismo identificador para definir una constante con un sentido y valor diferente. Si nuestro programa quiere hacer uso de ambas bibliotecas, ¿a cual de ellas nos referimos cuando usemos la constante `MAX`?. Necesitamos algún mecanismo para identificar exactamente a qué entidad nos referimos en cada caso.

El lenguaje de programación C++ permite solucionar este tipo de situaciones ambiguas mediante el uso de *espacios de nombres* (*namespace* en inglés), que permiten agrupar bajo una misma denominación (jerarquía) un conjunto de declaraciones y definiciones, de tal forma que dicha denominación será necesaria para identificar y diferenciar cada entidad declarada. En nuestro ejemplo, una de las constantes `MAX` sería definida en el ámbito de un espacio de nombres concreto y la otra en el ámbito de otro espacio de nombres diferente. Posteriormente podríamos referirnos a cada una de ellas indicando el identificador y el espacio de nombres en el que está definida, lo que evita cualquier posibilidad de ambigüedad.

Para definir un espacio de nombres se utiliza la palabra reservada **`namespace`** seguida por el *identificador* del espacio de nombres, y entre *llaves* las declaraciones y definiciones que deban estar bajo dicha jerarquía del espacio de nombres.

Los espacios de nombres pueden ser únicos para un determinado módulo o, por el contrario, pueden extenderse a múltiples módulos y bibliotecas gestionados por el mismo proveedor. Por ejemplo, todas las entidades definidas en la biblioteca estándar se encuentran bajo el espacio de nombres **`std`**.

El identificador del espacio de nombres puede ser derivado del propio nombre del fichero, puede incluir una denominación relativa al proveedor del módulo, o alguna otra denominación más compleja que garantice que no habrá colisiones en el identificador del espacio de nombres. Por ejemplo, podemos definir el módulo *complejo* dentro del espacio de nombres *umalcc*, que haría referencia a un proveedor del departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga.

```
main.cpp (Principal)
#include <iostream>
#include "complejo.hpp"
using namespace std ;
using namespace umalcc ;
// utilización de complejo
int main()
{
    ...
}
```

```
complejo.hpp (Interfaz)
#ifndef complejo_hpp_
#define complejo_hpp_
#include <...otros...>
// interfaz de complejo
namespace umalcc {
    ...
    ...
}
#endif
```

```
complejo.cpp (Implementación)
#include "complejo.hpp"
#include <...otros...>
// implementación de complejo
namespace umalcc {
    ...
    ...
    ...
    ...
}
```

Nótese que la inclusión de ficheros de encabezamiento se debe realizar **externamente** a la definición de los espacios de nombres.

Utilización de Espacios de Nombres

Una vez que las entidades han sido definidas dentro de un espacio de nombres, éstas no pueden ser utilizadas directamente, sino que es necesario algún tipo de *calificación* que permita referenciar e identificar a las entidades dentro de los espacios de nombres en los que han sido definidas. Ello se puede realizar de varias formas, dependiendo de las circunstancias donde se produzca esta utilización:

- Todos los identificadores definidos dentro de un espacio de nombres determinado son visibles y accesibles directamente desde dentro del mismo espacio de nombres, sin necesidad de calificación.

- Fuera del espacio de nombres, el uso de un identificador definido dentro de un espacio de nombres debe especificar claramente el espacio de nombres al que pertenece. Para ello se puede utilizar una calificación explícita, indicando el identificador que designa al espacio de

nombres, seguido del operador `::` y del identificador de la entidad a la que queremos aludir. Por ejemplo, para aludir a un identificador `MAX`, definido en el espacio de nombres `umalcc`, habría que utilizar `umalcc::MAX`.

- En el caso de ficheros de implementación (con extensión `.cpp`) podemos estar interesados en utilizar con frecuencia identificadores de un mismo espacio de nombres. En estos casos resultaría más cómodo poder omitir la calificación explícita. En estos casos resulta más cómodo suponer que, por defecto, nos referimos a un determinado espacio de nombres. Para ello se utiliza la directiva `using namespace`, que pone disponibles (accesibles) todos los identificadores de dicho espacio de nombres completo, que podrán ser accedidos directamente, sin necesidad de calificación explícita. Esto no resulta nuevo para nosotros, ya que estamos acostumbrados a utilizar dicha directiva para hacer accesible el espacio de nombres `std`

```
using namespace std ;
```

  Por ejemplo, como las bibliotecas estándar (`iostream`, `string` o `fstream`) se definen en el espacio de nombres `std`, para usar alguna de sus entidades (por ejemplo, el tipo `string`), tenemos dos posibilidades: utilizar una calificación explícita como se indica en el punto anterior (`std::string`) o, como hemos hecho hasta ahora en este curso, usar la directiva `using namespace` y suponer que los identificadores están definidos en el espacio de nombres `std`.

  El uso de la directiva `using namespace` no está recomendado en los ficheros de encabezamiento, donde **siempre utilizaremos** calificación explícita. Por ejemplo, si queremos crear un módulo con un fichero de encabezamiento *personas.hpp*, optaríamos por calificar explícitamente los identificadores cuando fuera necesario:

```
namespace umalcc {
    struct Persona {
        std::string nombre ;
        int edad ;
    } ;
    typedef std::array<int, 20> Vector ;
    void leer(std::string& nombre) ;
}
```

Es posible que se utilice la directiva `using namespace` para hacer accesible a varios espacios de nombres simultáneamente. En tal caso existe la posibilidad de que volvamos a tener problemas de *colisión* entre los identificadores. Puede haber varios identificadores que definen entidades diferentes y que son accesibles en dos espacios de nombres diferentes. Ello no supone ningún problema si nuestro programa no utiliza los identificadores en conflicto. Sin embargo, en caso de utilizarlos el compilador no podría saber a qué entidad nos estamos refiriendo. Para solucionar este tipo de conflictos el programador debe utilizar la *calificación explícita* con este identificador, eliminando así la ambigüedad en su utilización.

main.cpp
```
#include <iostream>
#include <string>
#include "datos.hpp"
using namespace std ;
using namespace umalcc ;
// utilización de datos
int main()
{
    string colision ;
    std::string nombre_1 ;
    umalcc::string nombre_2 ;
    ···
}
```

datos.hpp
```
#ifndef datos_hpp_
#define datos_hpp_
#include <array>
#include <...otros...>
// interfaz de datos
namespace umalcc {
    struct string {
        std::array<char, 50> datos ;
        int size ;
    } ;
    ···
}
#endif
```

datos.cpp
```
#include "datos.hpp"
#include <...otros...>
namespace umalcc {
    ···
    ···
    ···
    ···
    ···
    ···
    ···
    ···
    ···
}
```

Es importante remarcar que no es adecuado utilizar la directiva `using namespace` dentro de ficheros de encabezamiento. Ello es así porque estos ficheros están pensados para ser incluidos por

otros módulos, por lo que se pondrían disponibles (accesibles) todos los identificadores de dicho espacio de nombres en todos los ficheros que incluyan (`include`) dicho fichero de encabezamiento y ello podría provocar colisiones inesperadas y no deseadas.

Espacios de Nombres Anónimos

Los *espacios de nombres anónimos* permiten definir entidades privadas internas a los módulos de implementación, de tal forma que no puedan producir colisiones con las entidades públicas del sistema completo. De esta forma, cualquier declaración y definición realizada dentro de un espacio de nombres anónimo será únicamente visible en el módulo de implementación donde se encuentre (privada), pero no será visible en el exterior del módulo.

Adicionalmente, también es posible definir espacios de nombres anidados (dentro de otros espacios de nombres), pudiendo, de esta forma, definir jerarquías de espacios de nombres.

Ejemplo. Módulo Números Complejos

A continuación mostraremos todos los elementos estudiados en este capítulo, presentando un programa en el que hacemos diversas operaciones con números complejos. Es de suponer que, no solamente este programa, sino otros programas que podamos acometer en el futuro estarán interesados en trabajar con números complejos. Por ese motivo, diseñaremos nuestra solución mediante un módulo principal, en el que se realizan las pruebas de las operaciones deseadas, y un módulo *complejos*, que proporciona los tipos y operaciones necesarias para trabajar con números complejos. De este modo, otros programas que deseen trabajar con este tipo de números podrán reutilizar los tipos y operaciones que se describen en el interfaz del módulo sin necesidad de volver a desarrollarlo. Bastará con incluir el interfaz del módulo (*complejos.hpp*) y enlazar su código objeto (*complejos.o*).

El interfaz se define en un fichero de encabezamiento en el que nos aseguramos que no se produzca el problema de la doble inclusión. Utilizamos la directiva de compilación condicional y elegimos como guarda el identificador `_complejos_hpp_`. Además, definimos el espacio de nombres `umalcc` y nos aseguramos de que todas las entidades del interfaz están dentro de dicho espacio de nombres. Como consecuencia, cualquier uso de una de dichas entidades exigirá, bien una calificación explíta o bien el uso de la directiva `using namespace umalcc`.

El interfaz del módulo contiene la definición del tipo `Complejo` y la declaración de los prototipos de las operaciones que se pueden realizar con dicho tipo (`leer`, `sumar`, ...).

```
//- fichero: complejos.hpp ------------------------------------------
#ifndef complejos_hpp_
#define complejos_hpp_
namespace umalcc {
    //----------------------------------
    struct Complejo {
        double real ; // parte real del numero complejo
        double imag ; // parte imaginaria del numero complejo
    } ;
    //----------------------------------
    void sumar(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // sumar los numeros complejos (a) y (b).
    //----------------------------------
    void restar(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // restar los numeros complejos (a) y (b).
    //----------------------------------
    void multiplicar(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // multiplicar los numeros complejos (a) y (b).
```

```
    //----------------------------------
    void dividir(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // dividir los numeros complejos (a) y (b).
    //----------------------------------
    bool iguales(const Complejo& a, const Complejo& b) ;
    // Devuelve true si los numeros complejos (a) y (b) son iguales.
    //----------------------------------
    void escribir(const Complejo& a) ;
    // muestra en pantalla el numero complejo (a)
    //----------------------------------
    void leer(Complejo& a) ;
    // lee de teclado el valor del numero complejo (a).
    // lee la parte real y la parte imaginaria del numero
    //----------------------------------
}
#endif
//- fin: complejos.hpp ----------------------------------------------
```

La implementación del módulo deberá relizarse en un fichero independiente (*complejos.cpp*), que debe incluir el correspondiente fichero de encabezamiento para tener acceso a las definiciones y declaraciones allí realizadas. Además, como en este ejemplo necesitaremos efectuar entrada/salida con la consola, también incluimos la biblioteca estándar `iostream`. El uso de las entidades de dicha biblioteca exige utilizar el espacio de nombres `std`. En este caso, por comodidad optamos por usar la directiva `using namespace` lo que nos permitirá omitir en el resto del fichero el uso de calificación explícita (`std::`).

```
//- fichero: complejos.cpp ------------------------------------------
#include "complejos.hpp"
#include <iostream>
using namespace std ;
using namespace umalcc ;
```

En este ejemplo la implementación del módulo consiste en la implementación de los subprogramas cuyos prototipos fueron declarados en el fichero de encabezamiento. La implementación de dichos subprogramas puede utilizar otros subprogramas internos al módulo como, por ejemplo, las funciones `sq` o `abs`, que se definen dentro de un espacio de nombres anónimo, para hacerlas privadas al módulo.

```
namespace {
    //----------------------------------
    //-- Subprogramas Auxiliares -------
    //----------------------------------
    // cuadrado de un numero (a^2)
    inline double sq(double a)
    {
        return a*a ;
    }
    //----------------------------------
    // Valor absoluto de un numero
    inline double abs(double a)
    {
        if (a < 0) {
            a = -a ;
        }
        return a ;
    }
    //----------------------------------
    // Dos numeros reales son iguales si la distancia que los
```

```
    // separa es lo suficientemente pequenya
    inline bool igual(double a, double b)
    {
        return abs(a-b) <= 1e-6 ;
    }
}
```

Como las entidades del fichero de encabezamiento fueron definidas en el espacio de nombres `umalcc`, ahora procedemos a su implementación en dicho espacio de nombres. Los identificadores que dan nombre a dichas entidades son directamente accesibles en dicho espacio de nombres y, por tanto, pueden ser utilizados directamente, sin necesidad de calificación (por ejemplo, `Complejo`).

```
namespace umalcc {
    //----------------------------------
    //-- Implementación ----------------
    //----------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // sumar los numeros complejos (a) y (b).
    void sumar(Complejo& r, const Complejo& a, const Complejo& b)
    {
        r.real = a.real + b.real ;
        r.imag = a.imag + b.imag ;
    }
    //----------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // restar los numeros complejos (a) y (b).
    void restar(Complejo& r, const Complejo& a, const Complejo& b)
    {
        r.real = a.real - b.real ;
        r.imag = a.imag - b.imag ;
    }
    //----------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // multiplicar los numeros complejos (a) y (b).
    void multiplicar(Complejo& r, const Complejo& a, const Complejo& b)
    {
        r.real = (a.real * b.real) - (a.imag * b.imag) ;
        r.imag = (a.real * b.imag) + (a.imag * b.real) ;
    }
    //----------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // dividir los numeros complejos (a) y (b).
    void dividir(Complejo& r, const Complejo& a, const Complejo& b)
    {
        double divisor = sq(b.real) + sq(b.imag) ;
        if (igual(0.0, divisor)) {
            r.real = 0 ;
            r.imag = 0 ;
        } else {
            r.real = ((a.real * b.real) + (a.imag * b.imag)) / divisor ;
            r.imag = ((a.imag * b.real) - (a.real * b.imag)) / divisor ;
        }
    }
    //----------------------------------
    // Devuelve true si los numeros complejos (a) y (b) son iguales.
    bool iguales(const Complejo& a, const Complejo& b)
    {
        return igual(a.real, b.real) && igual(a.imag, b.imag) ;  }
    //----------------------------------
```

```
    // muestra en pantalla el numero complejo (a)
    void escribir(const Complejo& a)
    {
        cout << "{ " << a.real << ", " << a.imag << " }" ;
    }
    //----------------------------------
    // lee de teclado el valor del numero complejo (a).
    // lee la parte real y la parte imaginaria del numero
    void leer(Complejo& a)
    {
        cin >> a.real >> a.imag ;
    }
    //----------------------------------
}
//- fin: complejos.cpp ----------------------------------------------
```

Una vez que disponemos del módulo para operar con números complejos, a continuación mostramos un ejemplo de su utilización en el siguiente programa, que contiene algunos subprogramas para realizar pruebas de las operaciones básicas del mismo.

Nuestro módulo principal comienza incluyendo la biblioteca estándar `iostream`, porque en el mismo utilizaremos operaciones de entrada/salida con la consola. Además, declara variables de tipo `Complejo` y hace uso de diferentes subprogramas para trabajar con dicho tipo. Como consecuencia, debemos incluir el fichero de encabezamiento con el interfaz de dicho módulo (*complejos.hpp*), consiguiendo así que todas sus definiciones y declaraciones sean visibles en el módulo principal.

```
//- fichero: main.cpp ------------------------------------------
#include <iostream>
#include "complejos.hpp"
```

Sabemos que las entidades de la biblioteca `iostream` han sido definidas en el espacio de nombres `std` y las del módulo *complejos* han sido definidas en el espacio de nombres `umalcc`. Utilizamos la directiva `using namespace` para evitar la necesidad de utilizar calificación explícita cuando hagamos uso de las entidades definidas en las mismas. Sin embargo, hay una situación en la que nos vemos obligados a utilizar calificación explícita. Como se puede observar, el identificador `leer` tiene dos usos diferentes. Por una parte, es el nombre del subprograma utilizado para leer un número complejo, que ha sido declarado en el interfaz del módulo (*complejos.hpp*). Por otra parte, en el módulo principal tenemos otro subprograma que también se llama `leer`, tiene los mismos parámetros, pero no tiene ninguna relación con el primero. Si queremos utilizar el subprograma `leer` que ha sido definido en el interfaz del módulo, es necesario que utilicemos el identificador el espacio de nombres (`umalcc`) para calificar explícitamente el nombre del subprograma, como se puede ver a continuación. Nótese que, para invocar al subprograma `leer` del módulo principal, se debe calificar explícitamente con `::` directamente desde el espacio de nombres global.

```
using namespace std ;
using namespace umalcc ;
//-------------------------------------
void leer(Complejo& c)
{
    cout << "Introduzca un numero complejo { real img }: " ;
    umalcc::leer(c) ;
}
```

A continuación, se define el resto de subprogramas adecuados para proceder a las pruebas deseadas y la función `main`.

```
//-------------------------------------
void prueba_suma(const Complejo& c1, const Complejo& c2)
{
```

```
        Complejo c0 ;
        sumar(c0, c1, c2) ;
        escribir(c1) ;
        cout <<" + " ;
        escribir(c2) ;
        cout <<" = " ;
        escribir(c0) ;
        cout << endl ;
        Complejo aux ;
        restar(aux, c0, c2) ;
        if (! iguales(c1, aux)) {
            cout << "Error en operaciones de suma/resta"<< endl ;
        }
    }
    //-------------------------------------
    void prueba_resta(const Complejo& c1, const Complejo& c2)
    {
        Complejo c0 ;
        restar(c0, c1, c2) ;
        escribir(c1) ;
        cout <<" - " ;
        escribir(c2) ;
        cout <<" = " ;
        escribir(c0) ;
        cout << endl ;
        Complejo aux ;
        sumar(aux, c0, c2) ;
        if (! iguales(c1, aux)) {
            cout << "Error en operaciones de suma/resta"<< endl ;
        }
    }
    //-------------------------------------
    void prueba_mult(const Complejo& c1, const Complejo& c2)
    {
        Complejo c0 ;
        multiplicar(c0, c1, c2) ;
        escribir(c1) ;
        cout <<" * " ;
        escribir(c2) ;
        cout <<" = " ;
        escribir(c0) ;
        cout << endl ;
        Complejo aux ;
        dividir(aux, c0, c2) ;
        if (! iguales(c1, aux)) {
            cout << "Error en operaciones de mult/div"<< endl ;
        }
    }
    //-------------------------------------
    void prueba_div(const Complejo& c1, const Complejo& c2)
    {
        Complejo c0 ;
        dividir(c0, c1, c2) ;
        escribir(c1) ;
        cout <<" / " ;
        escribir(c2) ;
        cout <<" = " ;
        escribir(c0) ;
        cout << endl ;
```

```
    Complejo aux ;
    multiplicar(aux, c0, c2) ;
    if (! iguales(c1, aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
//-------------------------------------
int main()
{
    Complejo c1, c2 ;
    // calificación explícita del espacio de nombres global
    // para evitar colisión en invocación al subprograma
    // leer(Complejo& c) del espacio de nombres global
    ::leer(c1) ;
    ::leer(c2) ;
    //---------------------------------
    prueba_suma(c1, c2) ;
    prueba_resta(c1, c2) ;
    prueba_mult(c1, c2) ;
    prueba_div(c1, c2) ;
    //---------------------------------
}
//- fin: main.cpp ----------------------------------------------
```

Para finalizar, procederemos a su compilación separada y a su enlazado utilizando *GNU GCC*:

```
g++ -ansi -Wall -Werror -c complejos.cpp
g++ -ansi -Wall -Werror -c main.cpp
g++ -ansi -Wall -Werror -o main main.o complejos.o
```

aunque alternativamente, dicho proceso también se puede realizar en dos pasos:

```
g++ -ansi -Wall -Werror -c complejos.cpp
g++ -ansi -Wall -Werror -o main main.cpp complejos.o
```

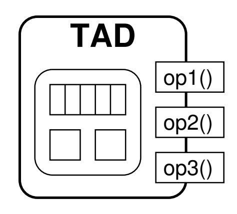
o incluso en un único paso:

```
g++ -ansi -Wall -Werror -o main main.cpp complejos.cpp
```

Capítulo 11

Tipos Abstractos de Datos

A medida que aumenta la complejidad del problema a resolver, del mismo modo deben aumentar los niveles de abstracción necesarios para diseñar y construir su solución algorítmica. Así, la abstracción procedimental permite aplicar adecuadamente técnicas de *diseño descendente* y *refinamientos sucesivos* en el desarrollo de algoritmos y programas. La programación modular permite aplicar la abstracción a mayor escala, permitiendo abstraer sobre conjuntos de operaciones y los datos sobre los que se aplican. De esta forma, a medida que aumenta la complejidad del problema a resolver, aumenta también la complejidad de las estructuras de datos necesarias para su resolución, y este hecho requiere, así mismo, la aplicación de la abstracción a las estructuras de datos.



La aplicación de la abstracción a las estructuras de datos da lugar a los *Tipos Abstractos de Datos* (TAD), donde se especifica el concepto que representa un determinado tipo de datos, y la semántica (el significado) de las operaciones que se le pueden aplicar, pero donde su representación e implementación internas permanecen ocultas e inaccesibles desde el exterior, de tal forma que no son necesarias para su utilización. Así, podemos considerar que un tipo abstracto de datos *encapsula* una determinada estructura abstracta de datos, impidiendo su manipulación directa, permitiendo solamente su manipulación a través de las operaciones especificadas. De este modo, los tipos abstractos de datos proporcionan un mecanismo adecuado para el diseño y reutilización de software fiable y robusto.

Para un determinado tipo abstracto de datos, se pueden distinguir tres niveles:

- Nivel de utilización, donde se utilizan objetos de un determinado tipo abstracto de datos, basándose en la especificación del mismo, de forma independiente a su implementación y representación concretas. Así, estos objetos se manipulan mediante la invocación a las operaciones especificadas en el TAD.

- Nivel de especificación, donde se especifica el tipo de datos, el concepto abstracto que representa y la semántica y restricciones de las operaciones que se le pueden aplicar. Este nivel representa el *interfaz* público del tipo abstracto de datos.

Utilización de TAD
Especificación de TAD
Implementación de TAD

- Nivel de implementación, donde se define e implementa tanto las estructuras de datos que soportan la abstracción, como las operaciones que actúan sobre ella según la semántica especificada. Este nivel interno permanece privado, y no es accesible desde el exterior del tipo abstracto de datos.

Nótese que para una determinada especificación de un tipo abstracto de datos, su implementación puede cambiar sin que ello afecte a la utilización del mismo.

11.1. Tipos Abstractos de Datos en C++: Clases

En el lenguaje de programación C++, las clases dan la posibilidad al programador de definir tipos abstractos de datos, de tal forma que permiten definir su representación interna (compuesta por sus atributos miembros), la forma en la que se crean y se destruyen, como se asignan y se pasan como parámetros, y las operaciones que se pueden aplicar (denominadas funciones miembro o simplemente métodos). De esta forma se hace el lenguaje extensible. Así mismo, la definición de tipos abstractos de datos mediante clases puede ser combinada con la definición de módulos (véase 10), haciendo de este modo posible la reutilización de estos nuevos tipos de datos.

Así, en C++ una determinada *clase* define un determinado *tipo abstracto de datos*, y un *objeto* se corresponde con una determinada *instancia de una clase*, de igual forma que una *variable* se corresponde con una determinada *instancia de un tipo de datos*. Así, las características de un determinado *objeto*, su estado y su comportamiento, están determinadas por la *clase* a la que pertenece. Del mismo modo, el *objeto* podrá ser manipulado e interactuar con otros objetos a través de los *métodos* definidos por la *Clase* a la que pertenece.

11.1.1. Métodos y Atributos

La clase representa una abstracción de datos, y los *métodos* definen su comportamiento. Los *métodos* son algoritmos *especiales* definidos por la *clase*, y se aplican sobre los objetos, manipulando el estado interno del objeto sobre el que se aplican.

La invocación a métodos puede llevar parámetros asociados, así como producir un resultado, además de manipular el estado interno del objeto sobre el que se aplica. Para invocar a un determinado método sobre un objeto, ese método debe estar definido por la clase a la que el objeto pertenece.

Los *atributos* estan protegidos, de tal forma que sólo se permite su acceso y manipulación a través de los métodos definidos por la clase. Los *atributos* almacenan los valores del estado interno de cada objeto, considerando que cada objeto tiene un estado interno asociado, independiente de los otros objetos.

11.1.2. Definición de Clases

Aunque C++ permite implementar las clases utilizando una definición *en línea*, en este capítulo nos centraremos en la implementación separada de los métodos de las clases. Además, combinaremos la especificación de la clase y su implementación con los conceptos de programación modular vistos en el capítulo anterior (véase 10), de tal forma que la definición de la clase se realizará en el fichero de cabecera (`hpp`) de un determinado módulo, y la implementación de la clase se realizará en el fichero de implementación (`cpp`) del módulo.

La definición de la clase se realizará en el fichero de cabecera (`hpp`) de un determinado módulo, dentro de las guardas y espacio de nombres adecuado. Para ello, se especifica la palabra reservada `class` seguida por el identificador de la nueva clase (tipo) que se está definiendo, y entre llaves la definición de los atributos (miembros) que lo componen y de los métodos (funciones miembros) que se le pueden aplicar directamente a los objetos de la clase. Finalmente el delimitador *punto y coma* (`;`) debe seguir al delimitador *cierra-llaves* (`}`).

```
//- fichero: complejos.hpp ------------------------------------------
#ifndef complejos_hpp_
#define complejos_hpp_
namespace umalcc {
    class Complejo {
        // ...
    } ;
}
#endif
//- fin: complejos.hpp ----------------------------------------------
```

Zona Privada y Zona Pública

En la definición de una clase, se pueden distinguir dos ámbitos de visibilidad (accesibilidad), la parte *privada*, cuyos miembros sólo serán accesibles desde un ámbito *interno* a la propia clase, y la parte *pública*, cuyos miembros son accesibles tanto desde un ámbito *interno* como desde un ámbito *externo* a la clase.

La parte privada comprende desde el principio de la definición de la clase hasta la etiqueta `public:`, y la parte pública comprende desde esta etiqueta hasta que se encuentra otra etiqueta `private:`. Cada vez que se especifica una de las palabras reservadas `public:` o `private:`, las declaraciones que la siguen adquieren el atributo de visibilidad dependiendo de la etiqueta especificada.

```
class Complejo {
public:
    // ... zona pública ...
private:
    // ... zona privada ...
} ;
```

Atributos

Los atributos componen la representación interna de la clase, y se definen usualmente en la zona de visibilidad privada de la clase, con objeto de proteger el acceso a dicha representación interna. Su definición se realiza de igual forma a los campos de los registros (véase 6.3).

De igual modo a los registros y sus campos, cada *objeto* que se defina del tipo de la clase, almacenará su propia representación interna de los atributos de forma independiente a los otros objetos (instancias de la misma clase).

```
class Complejo {
public:
    // ...
private:
    double real ; // parte real del numero complejo
    double imag ; // parte imaginaria del numero complejo
} ;
```

El Constructor por Defecto

El *constructor* de una clase permite construir e inicializar un *objeto*. El *constructor por defecto* es el mecanismo por defecto utilizado para construir objetos de este tipo cuando no se especifica ninguna forma explícita de construcción. Así, será el encargado de construir el objeto con los valores iniciales adecuados en el momento en que sea necesaria dicha construcción, por ejemplo cuando el flujo de ejecución alcanza la declaración de una variable de dicho tipo (véase 11.1.3).

Los constructores se declaran con el mismo identificador de la clase, seguidamente se especifican entre paréntesis los parámetros necesarios para la construcción, que en el caso del constructor por defecto, serán vacíos. Nótese que la definición del constructor no especifica ningún tipo de valor devuelto (ni siquiera `void`).

```
class Complejo {
public:
    Complejo() ;                                // Constructor por Defecto
    // ...
private:
    // ...
} ;
```

Métodos Generales y Métodos Constantes

Los métodos se corresponden con las operaciones que permiten manipular de muy diversa forma el estado interno de un determinado objeto como instancia de una determinada clase. Puede haber métodos definidos en el ámbito público de la clase, en cuyo caso podrán ser invocados tanto desde métodos internos de la clase, como desde el exterior de la clase, y métodos definidos en el ámbito privado de la clase, en cuyo caso sólo podrán ser invocados desde métodos internos de la clase. Estos métodos definidos en el ámbito privado de la clase suelen ser definidos como métodos auxiliares que facilitan la implementación de otros métodos más complejos.

Los métodos se declaran como los prototipos de los subprogramas (véase 5.7), pero teniendo en cuenta que son aplicados a un objeto instancia de la clase a la que pertenece, y que, por lo tanto, no es necesario que dicho objeto sea recibido como parámetro.

Los métodos de una clase pueden tener el calificador `const` especificado después de los parámetros, en cuyo caso indica que el método no modifica el estado interno del objeto (atributos), por lo que se puede aplicar tanto a objetos constantes como variables. En otro caso, si dicho calificador `const` no aparece, entonces significa que el método sí modifica el estado interno del objeto (atributos), por lo que sólo podrá ser aplicado a objetos variables, y por el contrario no podrá ser aplicado a objetos constantes.

```
class Complejo {
public:
    // ...
    //----------------------------
    void sumar(const Complejo& a, const Complejo& b) ;
    // asigna al numero complejo (actual) el resultado de
    // sumar los numeros complejos (a) y (b).
    //----------------------------
    bool igual(const Complejo& b) const ;
    // Devuelve true si el numero complejo (actual) es
    // igual al numero complejo (b)
    //----------------------------
    void escribir() const ;
    // muestra en pantalla el numero complejo (actual)
    //----------------------------
    void leer() ;
    // lee de teclado el valor del numero complejo (actual).
    // lee la parte real y la parte imaginaria del numero
    //------------------------------
    // ...
private:
    // ...
} ;
```

11.1.3. Utilización de Clases

Un tipo abstracto de datos, definido como una clase encapsulada dentro de un módulo (en un determinado espacio de nombres), puede ser utilizado por cualquier otro módulo que lo necesite. Para ello, deberá incluir el fichero de cabecera donde se encuentra la definición de la clase, y podrá definir tantos objetos (instancias de dicha clase) como sean necesarios. Para ello, deberá utilizar calificación explícita o implícita del especio de nombres dependiendo del contexto de su utilización (ficheros de encabezamiento o de implementación respectivamente).

Instancias de Clase: Objetos

Un objeto es una instancia de una clase, y podremos definir tantos objetos cuyo tipo sea de una determinada clase como sea necesario, de tal modo que cada objeto contiene su propia representación interna (atributos) de forma independiente del resto.

La definición de un objeto de una determinada clase se realiza de igual forma a la definición de una variable (o constante) de un determinado tipo, de tal forma que cada objeto será una instancia independiente de una determinada clase (tipo abstracto de datos).

```
//- fichero: main.cpp ---------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2 ;
    // ...
}
//- fin: main.cpp -------------------------------------------------
```

	c1	c2
real:	0.0	0.0
imag:	0.0	0.0

Es importante remarcar que cada objeto, definido de una determinada clase, es una instancia independiente de los otros objetos definidos de la misma clase, con su propia memoria para contener de forma independiente el estado de su representación interna.

Tiempo de Vida de los Objetos

Durante la ejecución del programa, cuando el flujo de ejecución llega a la sentencia donde se define un determinado objeto, entonces se reserva espacio en memoria para contener a dicho objeto, y se invoca al constructor especificado (si no se especifica ningún constructor, entonces se invoca al constructor por defecto) para construir adecuadamente al objeto, siendo de esta forma accesible desde este punto de construcción hasta que el flujo de ejecución alcanza el final de bloque donde el objeto ha sido definido, en cuyo caso el objeto se destruye (invocando a su *destructor*) y se libera la memoria que ocupaba, pasando de este modo a estar inaccesible.

```
//- fichero: main.cpp ---------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;                   // construcción de c1 (1 vez)
    for (int i = 0 ; i < 3 ; ++i) {
        Complejo c2 ;               // construcción de c2 (3 veces)
        // ...
    }                               // destrucción de c2 (3 veces)
    // ...
}                                   // destrucción de c1 (1 vez)
//- fin: main.cpp -------------------------------------------------
```

Manipulación de los Objetos

Una vez que un objeto es accesible, se puede manipular invocando a los métodos **públicos** definidos en su interfaz. Esta invocación de los métodos se aplica sobre un determinado objeto en concreto, y se realiza especificando el identificador del objeto sobre el que recae la invocación al método, seguido por el símbolo punto (.) y por la invocación al método en cuestión, es decir, el identificador del método y los parámetros actuales necesarios entre paréntesis. Hay que tener en cuenta que a los objetos constantes sólo se les podrán aplicar métodos constantes. Así mismo, nótese que no es posible acceder a los atributos ni métodos que hayan sido definidos en la parte privada de la clase.

```
//- fichero: main.cpp ----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2, c3 ;           // construcción de c1, c2, c3
    c1.leer() ;
    c2.leer() ;
    c3.sumar(c1, c2) ;
    c3.escribir() ;
}                                  // destrucción de c1, c2, c3
//- fin: main.cpp --------------------------------------------------
```

	c1	c2	c3
real:	5.3	2.5	7.8
imag:	2.4	7.3	9.7

Paso de Parámetros de Objetos

Es importante considerar que las clases (tipos abstractos de datos) son *tipos compuestos*, y por lo tanto deben seguir las mismas convenciones para el paso de parámetros de tipos compuestos (véase 6.1), es decir, los parámetros de salida o entrada/salida se pasan por *referencia*, y los parámetros de entrada se pasan por *referencia constante*.

11.1.4. Implementación de Clases

La implementación de los métodos de la clase se realizará en el fichero de implementación (`cpp`) del módulo correspondiente, dentro del mismo espacio de nombres en el que fue realizada la definición de la clase en el fichero de cabecera.

En el fichero de implementación se podrán definir, dentro del espacio de nombres adecuado, las constantes, tipos y subprogramas auxiliares necesarios que nos faciliten la implementación de los métodos de la clase.

Para implementar un determinado constructor o método de la clase, dentro del mismo espacio de nombres que la definición de la clase, se *calificará explícitamente* el identifidor del método con el identificador de la clase a la que pertenece.

```
//- fichero: complejos.cpp -----------------------------------------
#include "complejos.hpp"
namespace umalcc {
    // ...
    //----------------------------
    Complejo::Complejo()                        // Constructor por Defecto
        // ...
    //----------------------------
    void Complejo::sumar(const Complejo& a, const Complejo& b)
        // ...
    //----------------------------
    bool Complejo::igual(const Complejo& b) const
        // ...
    //----------------------------
    void Complejo::escribir() const
        // ...
    //----------------------------
    void Complejo::leer()
        // ...
    //----------------------------
    // ...
}
//- fin: complejos.cpp ---------------------------------------------
```

Métodos

En la implementación de un determinado método de una clase, éste método puede invocar directamente a cualquier otro método de la clase sin necesidad de aplicar el operador punto (`.`). Así mismo, un método de la clase puede *acceder directamente a los atributos del objeto* sobre el que se invoque dicho método, sin necesidad de aplicar el operador punto (`.`), ni necesidad de recibirlo como parámetro. Por ejemplo:

```
void Complejo::sumar(const Complejo& a, const Complejo& b)
{
    real = a.real + b.real ;
    imag = a.imag + b.imag ;
}
```

Sin embargo, para acceder a los atributos de los objetos recibidos como parámetros, si son accesibles desde la implementación de una determinada clase, es necesario especificar el objeto (mediante su identificador) seguido por el operador punto (`.`) y a continuación el identificador del atributo en cuestión.

Así, podemos ver como para calcular la suma de números complejos, se asigna a las partes real e imaginaria del número complejo que estamos calculando (el objeto sobre el que se aplica el método `sumar`) la suma de las partes real e imaginaria respectivamente de los números complejos que recibe como parámetros. Por ejemplo, cuando se ejecuta la sentencia:

```
c3.sumar(c1, c2) ;
```

la sentencia correspondiente a la implementación del método `sumar(...)`:

```
    real = a.real + b.real ;
```

almacenará en el atributo `real` del número complejo `c3` el resultado de sumar los valores del atributo `real` de los números complejos `c1` y `c2`. De igual modo sucederá con el atributo `imag` del número complejo `c3`, que almacenará el resultado de sumar los valores del atributo `imag` de los números complejos `c1` y `c2`.

Constructores

En la implementación de los constructores de la clase, también será calificado explícitamente con el identificador de la clase correspondiente. Después de la definición de los parámetros, a continuación del delimitador (`:`), se especifica la *lista de inicialización*, donde aparecen, separados por comas y según el orden de declaración, todos los atributos miembros del objeto, así como los valores con los que serán inicializados especificados entre paréntesis (se invoca al constructor adecuado según los parámetros especificados entre paréntesis, de tal forma que los paréntesis vacíos representan la construcción por defecto). A continuación se especifican entre llaves las sentencias pertenecientes al cuerpo del constructor para realizar las acciones adicionales necesarias para la construcción del objeto. Si no es necesario realizar ninguna acción adicional, entonces el cuerpo del constructor se dejará vacío. Nótese que la definición del constructor no especifica ningún tipo de valor devuelto (ni siquiera `void`).

Por ejemplo, implementaremos el constructor por defecto de la clase `Complejo` para que inicialice cada atributo (parte real e imaginaria) del objeto que se construya invocando a su propio constructor por defecto respectivamente (según su tipo):

```
Complejo::Complejo()                        // Constructor por Defecto
    : real(), imag() { }
```

Alternativamente, la siguiente implementación es equivalente, para que inicialice cada atributo (parte real e imaginaria) del objeto que se construya con el valor cero (0.0) especificado entre paréntesis.

```
Complejo::Complejo()                        // Constructor por Defecto
    : real(0.0), imag(0.0) { }
```

11.1.5. Ejemplo

Por ejemplo, el *TAD número complejo* representa el siguiente concepto matemático de número complejo:

> Un número complejo representa un punto en el plano complejo, compuesto por dos componentes que representan la parte real y la parte imaginaria del número (abcisa y ordenada respectivamente en el plano cartesiano), al cual se le pueden aplicar las operaciones de suma, resta, multiplicación y división, así como la comparación de igualdad.

Definición

```
//- fichero: complejos.hpp -----------------------------------------
#ifndef complejos_hpp_
#define complejos_hpp_
namespace umalcc {
    //----------------------------------
    const double ERROR_PRECISION = 1e-6 ;
    //----------------------------------
    class Complejo {
    public:
        //----------------------------------------------------------
        //-- Métodos Públicos --------------------------------------
        //----------------------------------------------------------
        Complejo() ;                            // Constructor por Defecto
        //----------------------------
        double parte_real() const ;
        // devuelve la parte real del numero complejo
        //----------------------------
        double parte_imag() const ;
        // devuelve la parte imaginaria del numero complejo
        //----------------------------
        void sumar(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // sumar los numeros complejos (a) y (b).
        //----------------------------
        void restar(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // restar los numeros complejos (a) y (b).
        //----------------------------
        void multiplicar(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // multiplicar los numeros complejos (a) y (b).
        //----------------------------
        void dividir(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // dividir los numeros complejos (a) y (b).
        //----------------------------
        bool igual(const Complejo& b) const ;
        // Devuelve true si el numero complejo (actual) es
        // igual al numero complejo (b)
        //----------------------------
        void escribir() const ;
        // muestra en pantalla el numero complejo (actual)
        //----------------------------
        void leer() ;
        // lee de teclado el valor del numero complejo (actual).
        // lee la parte real y la parte imaginaria del numero
```

```
        //-----------------------------
    private:
        //----------------------------------------------------------
        //-- Atributos Privados ------------------------------------
        //----------------------------------------------------------
        double real ; // parte real del numero complejo
        double imag ; // parte imaginaria del numero complejo
        //----------------------------------------------------------
    } ;
}
#endif
//- fin: complejos.hpp ---------------------------------------------
```

Implementación

```
//- fichero: complejos.cpp -----------------------------------------
#include "complejos.hpp"
#include <iostream>
using namespace std ;
using namespace umalcc ;
//------------------------------------------------------------------------------
// Espacio de nombres anonimo. Es una parte privada de la
// implementacion. No es accesible desde fuera del modulo
//------------------------------------------------------------------------------
namespace {
    //----------------------------------------------------------
    //-- Subprogramas Auxiliares -------------------------------
    //----------------------------------------------------------
    // cuadrado de un numero (a^2)
    inline double sq(double a)
    {
        return a*a ;
    }
    //-------------------------
    // Valor absoluto de un numero
    inline double abs(double a)
    {
        return (a >= 0) ? a : -a ;
    }
    //-------------------------
    // Dos numeros reales son iguales si la distancia que los
    // separa es lo suficientemente pequenya
    inline bool iguales(double a, double b)
    {
        return abs(a-b) <= ERROR_PRECISION ;
    }
}
//------------------------------------------------------------------------------
// Espacio de nombres umalcc.
// Aqui reside la implementacion de la parte publica del modulo
//------------------------------------------------------------------------------
namespace umalcc {
    //----------------------------------------------------------
    //-- Métodos Públicos --------------------------------------
    //----------------------------------------------------------
    Complejo::Complejo()                        // Constructor por Defecto
        : real(0.0), imag(0.0) { }
    //----------------------------
    // devuelve la parte real del numero complejo
```

```
double Complejo::parte_real() const
{
    return real ;
}
//----------------------------
// devuelve la parte imaginaria del numero complejo
double Complejo::parte_imag() const
{
    return imag ;
}
//----------------------------
// asigna al numero complejo (actual) el resultado de
// sumar los numeros complejos (a) y (b).
void Complejo::sumar(const Complejo& a, const Complejo& b)
{
    real = a.real + b.real ;
    imag = a.imag + b.imag ;
}
//----------------------------
// asigna al numero complejo (actual) el resultado de
// restar los numeros complejos (a) y (b).
void Complejo::restar(const Complejo& a, const Complejo& b)
{
    real = a.real - b.real ;
    imag = a.imag - b.imag ;
}
//----------------------------
// asigna al numero complejo (actual) el resultado de
// multiplicar los numeros complejos (a) y (b).
void Complejo::multiplicar(const Complejo& a, const Complejo& b)
{
    real = (a.real * b.real) - (a.imag * b.imag) ;
    imag = (a.real * b.imag) + (a.imag * b.real) ;
}
//----------------------------
// asigna al numero complejo (actual) el resultado de
// dividir los numeros complejos (a) y (b).
void Complejo::dividir(const Complejo& a, const Complejo& b)
{
    double divisor = sq(b.real) + sq(b.imag) ;
    if (iguales(0.0, divisor)) {
        real = 0.0 ;
        imag = 0.0 ;
    } else {
        real = ((a.real * b.real) + (a.imag * b.imag)) / divisor ;
        imag = ((a.imag * b.real) - (a.real * b.imag)) / divisor ;
    }
}
//----------------------------
// Devuelve true si el numero complejo (actual) es
// igual al numero complejo (b)
bool Complejo::igual(const Complejo& b) const
{
    return iguales(real, b.real) && iguales(imag, b.imag) ;
}
//----------------------------
// muestra en pantalla el numero complejo (actual)
void Complejo::escribir() const
{
```

```
        cout << "{ " << real << ", " << imag << " }" ;
    }
    //----------------------------
    // lee de teclado el valor del numero complejo (actual).
    // lee la parte real y la parte imaginaria del numero
    void Complejo::leer()
    {
        cin >> real >> imag ;
    }
    //----------------------------
}
//- fin: complejos.cpp ---------------------------------------------
```

Utilización

```
//- fichero: main.cpp ----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;
//------------------------------------
void leer(Complejo& c)
{
    cout << "Introduzca un numero complejo { real img }: " ;
    c.leer() ;
}
//------------------------------------
void prueba_suma(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.sumar(c1, c2) ;
    c1.escribir() ;
    cout <<" + " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.restar(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de suma/resta"<< endl ;
    }
}
//------------------------------------
void prueba_resta(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.restar(c1, c2) ;
    c1.escribir() ;
    cout <<" - " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.sumar(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de suma/resta"<< endl ;
    }
```

```
}
//------------------------------------
void prueba_mult(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.multiplicar(c1, c2) ;
    c1.escribir() ;
    cout <<" * " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.dividir(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
//------------------------------------
void prueba_div(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.dividir(c1, c2) ;
    c1.escribir() ;
    cout <<" / " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.multiplicar(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
//------------------------------------
int main()
{
    Complejo c1, c2 ;
    leer(c1) ;
    leer(c2) ;
    //--------------------------------
    prueba_suma(c1, c2) ;
    prueba_resta(c1, c2) ;
    prueba_mult(c1, c2) ;
    prueba_div(c1, c2) ;
    //--------------------------------
}
//- fin: main.cpp ---------------------------------------------------
```

11.2. Tipos Abstractos de Datos en C++: Más sobre Clases

Constantes de Ámbito de Clase

Las constantes de ámbito de clase se definen especificando los calificadores `static const`, seguidos por el tipo, el identificador y el valor de la constante. Estas constantes serán comunes y accesibles a todas las instancias (objetos) de la clase. Por ejemplo, para definir la constante `MAX` con un valor de 256 en la zona privada de la clase:

```
    class ListaInt {
    public:
        // ...
    private:
        static const int MAX = 256 ;
        // ...
    } ;
```

Usualmente las constantes se definen en la zona privada de la clase, por lo que usualmente sólo serán accesibles internamente desde dentro de la clase. Sin embargo, en algunas situaciones puede ser conveniente definir la constante en la zona pública de la clase, entonces en este caso la constante podrá ser accedida desde el exterior de la clase, y será utilizada mediante calificación explícita utilizando el identificador de la clase. Por ejemplo:

```
    class ListaInt {
    public:
        static const int MAX = 256 ;
        // ...
    private:
        // ...
    } ;
    // ...
int main()
{
    int x = ListaInt::MAX ;
    // ...
}
```

Sin embargo, la definición de constantes de ámbito de clase de tipos diferentes a los integrales (`char`, `short`, `int`, `unsigned`, `long`), por ejemplo `float` y `double` es un poco más compleja, por lo que usualmente se realizará externamente a la definición de la clase, dentro del ámbito del módulo (en el fichero `hpp`, dentro del espacio de nombres del módulo, si debe ser pública, y en el fichero `cpp`, dentro del espacio de nombres anónimo, si debe ser privada).

Tipos de Ámbito de Clase

También se pueden definir tipos internos de ámbito de clase de igual forma a como se hace externamente a la clase, pero en este caso su ámbito de visibilidad estará restringido a la clase donde se defina. Estos tipos serán útiles en la definición de los atributos miembros de la clase, o para definir elementos auxiliares en la implementación del tipo abstracto de datos. Por ejemplo, para definir un tipo `Datos` como un array de 256 números enteros:

```
#include <array>
// ...
    class ListaInt {
    public:
        // ...
    private:
        static const int MAX = 256 ;
        typedef std::array<int, MAX> Datos ;
        struct Elemento {
            // ...
        } ;
        // ...
    } ;
```

Usualmente los tipos se definen en la zona privada de la clase, por lo que usualmente sólo serán accesibles internamente desde dentro de la clase. Sin embargo, en algunas situaciones puede ser conveniente definir el tipo en la zona pública de la clase, entonces en este caso el tipo podrá ser

accedido desde el exterior de la clase, y será utilizado mediante calificación explícita utilizando el identificador de la clase. Por ejemplo:

```
#include <array>
// ...
    class ListaInt {
    public:
        static const int MAX = 256 ;
        typedef std::array<int, MAX> Datos ;
        // ...
    private:
        // ...
    } ;
    // ...
int main()
{
    ListaInt::Datos d ;
    // ...
}
```

Nótese que los tipos deben ser públicos si forman parte de los parámetros de los métodos públicos, de tal forma que puedan ser utilizados externamente, allá donde sea necesario invocar a dichos métodos públicos.

Constructores Específicos

Los *constructores* de una clase permiten construir e inicializar un *objeto*. Anteriormente se ha explicado el *constructor por defecto*, el cual se invoca cuando se crea un determinado objeto y no se especifica que tipo de construcción se debe realizar. C++ permite, además, la definición e implementación de tantos constructores específicos como sean necesarios. Para ello, se debe especificar en la lista de parámetros, aquellos que sean necesarios para poder construir el objeto adecuadamente en cada circunstancia específica, de tal forma que será la lista de parámetros formales la que permita discriminar que constructor será invocado dependiendo de los parámetros actuales utilizados en la invocación al constructor. Nótese que la definición del constructor no especifica ningún tipo de valor devuelto (ni siquiera `void`).

Por ejemplo, podemos definir un constructor específico para que reciba dos números reales como parámetros (parte real e imaginaria respectivamente de un número complejo), los cuales serán utilizados para dar los valores iniciales a cada atributo correspondiente del objeto que se construya. Así, su definición podría ser:

```
class Complejo {
public:
    Complejo(double p_real, double p_imag) ;  // Constructor Específico
} ;
```

A continuación se puede ver como sería la implementación de este constructor específico, donde se inicializa el valor de cada atributo con el valor de cada parámetro recibido en la invocación de la construcción del objeto.

```
Complejo::Complejo(double p_real, double p_imag)  // Constructor específico
    : real(p_real), imag(p_imag) { }
```

Finalmente, a continuación podemos ver un ejemplo de como sería una posible invocación a dicho constructor específico (para `c2`), junto a una invocación al constructor por defecto (para `c1`):

```
//- fichero: main.cpp ---------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;
    Complejo c2(2.5, 7.3) ;
    // ...
}
//- fin: main.cpp -------------------------------------------------
```

real: 0.0
imag: 0.0
c1

real: 2.5
imag: 7.3
c2

Constructor por Defecto

Como se explicó anteriormente (véase 11.1.2 y 11.1.4), el *constructor por defecto* es el mecanismo por defecto utilizado para construir objetos de este tipo cuando no se especifica ninguna forma explícita de construcción. Así, será invocado automáticamente cuando se deba construir un determinado objeto, sin especificar explícitamente el tipo de construcción requerido, en el momento en que sea necesaria dicha construcción, por ejemplo cuando el flujo de ejecución alcanza la declaración de una variable de dicho tipo (véase 11.1.3).

El constructor por defecto es un método especial de la clase, ya que si el programador no define **ningún** constructor para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho constructor con el comportamiento por defecto de invocar automáticamente al constructor por defecto para cada atributo de *tipo compuesto* miembro de la clase. Nótese, sin embargo, que en el caso atributos de *tipo simple*, la implementación automática del compilador los dejará sin inicializar.

No obstante, el programador puede definir el constructor por defecto para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, la definición del constructor por defecto se corresponde con la definición de un constructor que no recibe ningún parámetro, y la implementación dependerá de las acciones necesarias para inicializar por defecto el estado interno del objeto que se está creando. Por ejemplo, para la clase `Complejo`:

```
class Complejo {
public:
    Complejo() ;              // Constructor por Defecto
} ;
```

A continuación se puede ver como sería la implementación del constructor por defecto:

```
Complejo::Complejo()          // Constructor por Defecto
    : real(0.0), imag(0.0) { }
```

Otra posible implementación podría ser la siguiente, que invoca explícitamente al constructor por defecto para cada atributo miembro de la clase (que en este caso se incializará a cero):

```
Complejo::Complejo()          // Constructor por Defecto
    : real(), imag() { }
```

Finalmente, a continuación podemos ver un ejemplo de como sería una invocación a dicho constructor por defecto:

```
//- fichero: main.cpp ---------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2 ;
    // ...
}
//- fin: main.cpp -------------------------------------------------
```

real: 0.0
imag: 0.0
c1

real: 0.0
imag: 0.0
c2

Constructor de Copia

El constructor de copia es el constructor que permite inicializar un determinado objeto como una copia de otro objeto de su misma clase. Así, se invoca automáticamente al inicializar el contenido de un objeto con el valor de otro objeto de su misma clase, y también es invocado automáticamente cuando un objeto de dicho tipo se pasa como *parámetro por valor* a subprogramas, aunque esto último, como se ha explicado previamente, está desaconsejado, ya que lo usual es pasar los tipos compuestos por referencia o por referencia constante.

El constructor de copia es un método especial de la clase, ya que si el programador no define dicho constructor de copia para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho constructor de copia con el comportamiento por defecto de invocar automáticamente al constructor de copia para cada atributo miembro de la clase, en este caso, tanto para atributos de *tipo simple* como de *tipo compuesto*.

No obstante, el programador puede definir el constructor de copia para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, la definición del constructor de copia se corresponde con la definición de un constructor que recibe como único parámetro *por referencia constante* un objeto del mismo tipo que la clase del constructor, y la implementación dependerá de las acciones necesarias para copiar el estado interno del objeto recibido como parámetro al objeto que se está creando. Por ejemplo, para la clase `Complejo`:

```
class Complejo {
public:
    Complejo(const Complejo& c) ;  // Constructor de Copia
} ;
```

y su implementación podría ser la siguiente, que en este caso coincide con la implementación que generaría automáticamente el compilador en caso de que no fuese implementado por el programador:

```
Complejo::Complejo(const Complejo& o)  // Constructor de Copia
    : real(o.real), imag(o.imag) { }   // Implementación automática
```

Finalmente, a continuación podemos ver un ejemplo de como sería una invocación al constructor de copia (para `c3` y `c4`), junto a una invocación a un constructor específico (para `c2`) y una invocación al constructor por defecto (para `c1`), así como la construcción por copia (para `c5` y `c6`) de objetos construidos invocando explícitamente a los constructores adecuados:

```
//- fichero: main.cpp ----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;             // Construcción por defecto
    Complejo c2(2.5, 7.3) ;   // Construcción específica
    Complejo c3(c1) ;         // Construcción de copia (de c1)
    Complejo c4 = c2 ;        // Construcción de copia (de c2)
    Complejo c5 = Complejo(); // Construcción de copia de Complejo por Defecto
    Complejo c6 = Complejo(3.1, 4.2); // Construcción de copia de Complejo Específico
    // ...
}
//- fin: main.cpp ---------------------------------------------------
```

	c1	*c2*	*c3*	*c4*	*c5*	*c6*
real:	0.0	2.5	0.0	2.5	0.0	3.1
imag:	0.0	7.3	0.0	7.3	0.0	4.2

Destructor

El *destructor* de una clase será invocado automáticamente (sin parámetros actuales) para una determinada instancia (objeto) de esta clase cuando dicho objeto deba ser destruido, normalmente ésto sucederá cuando el flujo de ejecución del programa salga del ámbito de visibilidad de dicho objeto (véase 11.1.3).

El destructor es un método especial de la clase, ya que si el programador no define dicho destructor para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho destructor con el comportamiento por defecto de invocar automáticamente al destructor para cada atributo de *tipo compuesto* miembro de la clase.

No obstante, el programador puede definir el destructor para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, el destructor de la clase se define mediante el símbolo ~ seguido del identificador de la clase y una lista de parámetros vacía, y la implementación dependerá de las acciones necesarias para destruir y liberar los recursos asociados al estado interno del objeto que se está destruyendo. Nótese que la definición del constructor no especifica ningún tipo de valor devuelto (ni siquiera `void`).

Posteriormente, el destructor invoca automáticamente a los destructores de los atributos miembros del objeto para que éstos sean destruidos. Por ejemplo, para la clase `Complejo`:

```
class Complejo {
public:
    ~Complejo() ;           // Destructor
} ;
```

y su implementación podría ser la siguiente, que en este caso coincide con la implementación que generaría automáticamente el compilador en caso de que no fuese implementado por el programador:

```
Complejo::~Complejo() { }   // Destructor: Implementación automática
```

Finalmente, a continuación podemos ver un ejemplo de como se invoca automáticamente al destructor de los objetos cuando termina su tiempo de vida (para `c1`, `c2`, `c3` y `c4`):

```
//- fichero: main.cpp ---------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;               // Construcción por defecto
    Complejo c2(2.5, 7.3) ;     // Construcción específica
    Complejo c3(c1) ;           // Construcción de copia (de c1)
    Complejo c4 = c2 ;          // Construcción de copia (de c2)
    // ...
}                               // Destrucción automática de c4, c3, c2 y c1
//- fin: main.cpp -------------------------------------------------
```

	c1	c2	c3	c4
real:	0.0	2.5	0.0	2.5
imag:	0.0	7.3	0.0	7.3

Operador de Asignación

El operador de asignación define como se realiza la asignación (=) para objetos de esta clase. No se debe confundir el operador de asignación con el constructor de copia, ya que el constructor de copia construye un nuevo objeto que no tiene previamente ningún valor, mientras que en el caso del operador de asignación, el objeto ya tiene previamente un valor que deberá ser sustituido por el nuevo valor. Este valor previo deberá, en ocasiones, ser destruido antes de realizar la asignación del nuevo valor.

El operador de asignación (=) es un método especial de la clase, ya que si el programador no define dicho operador de asignación para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho operador de asignación con el comportamiento por defecto de invocar automáticamente al operador de asignación para cada atributo miembro de la clase, tanto para atributos de *tipo simple* como de *tipo compuesto*.

No obstante, el programador puede definir el operador de asignación para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, la definición del operador de asignación se corresponde con la definición de un operador = que recibe como único parámetro *por referencia constante* un objeto del mismo tipo que la clase del constructor, devuelve una referencia al propio objeto que recibe la asignación, y la implementación dependerá de las acciones necesarias para destruir el estado interno del objeto que recibe la asignación y para asignar el estado interno del objeto recibido como parámetro al objeto que se está creando. Por ejemplo, para la clase `Complejo`:

```
class Complejo {
public:
    Complejo& operator=(const Complejo& o) ;  // Operador de Asignación
} ;
```

y su implementación podría ser la siguiente, que en este caso coincide con la implementación que generaría automáticamente el compilador en caso de que no fuese implementado por el programador:

```
Complejo& Complejo::operator=(const Complejo& o)  // Operador de Asignacion
{                                                  // Implementación automática
    real = o.real ;
    imag = o.imag ;
    return *this ;
}
```

El operador de asignación debe devolver el objeto actual (`return *this`) sobre el que recae la asignación.

Finalmente, a continuación podemos ver un ejemplo de como sería una invocación al operador de asignación (para `c3` y `c4`), junto a una invocación a un constructor específico (para `c2`) y una invocación al constructor por defecto (para `c1`), así como la asignación (para `c5` y `c6`) de objetos construidos invocando explícitamente a los constructores adecuados:

```
//- fichero: main.cpp ----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c3, c4, c5, c6; // Construcción por defecto de c1, c3, c4
    Complejo c2(2.5, 7.3) ;      // Construcción específica
    c3 = c1 ;                    // Asignación de c1 a c3
    c4 = c2 ;                    // Asignación de c2 a c4
    c5 = Complejo();             // Asignación de Complejo por Defecto
    c6 = Complejo(3.1, 4.2);     // Asignación de Complejo Específico
   // ...
}
//- fin: main.cpp --------------------------------------------------
```

	c1	*c2*	*c3*	*c4*	*c5*	*c6*
real:	0.0	2.5	0.0	2.5	0.0	3.1
imag:	0.0	7.3	0.0	7.3	0.0	4.2

Hay situaciones en las que los objetos que se asignan tienen representaciones internas complejas, y en estos casos puede ser necesario destruir el estado interno del objeto que recibe la asignación antes de asignar el nuevo valor. En este caso, es conveniente comprobar que no se está produciendo una *auto-asignación* del mismo objeto (`x = x`), ya que en este caso se destruiría la representación interna del objeto antes de haberla asignado, con los errores que ello trae asociado. Por lo tanto, suele ser habitual que el operador de asignación implemente una condición para evitar la asignación en el caso de que se produzca una *auto-asignación*, de la siguiente forma:

```
Complejo& Complejo::operator=(const Complejo& o)  // Operador de Asignacion
{
    if (this != &o) {
        // destruir el valor anterior (en este caso no es necesario)
        real = o.real ;
        imag = o.imag ;
    }
    return *this ;
}
```

Así, `this` representa la dirección en memoria del objeto que recibe la asignación, y `&o` representa la dirección en memoria del objeto que se recibe como parámetro. Si ambas direcciones son diferentes, entonces significa que son variables diferentes y se puede realizar la asignación.

11.2.1. Ejemplo

Veamos un Tipo Abstracto de Datos `Lista` de enteros, la cual permite almacenar una secuencia de número enteros, permitiendo insertar, eliminar, acceder y modificar elementos según la posición que ocupen en la secuencia de números.

Definición

```
//- fichero: lista.hpp ------------------------------------------
#ifndef lista_hpp_
#define lista_hpp_
#include <array>
namespace umalcc {
    class ListaInt {
    public:
        //----------------------------------------------------------
        //-- Métodos Públicos --------------------------------------
        //----------------------------------------------------------
        // ~ListaInt() ;                    // Destructor Automático
        //------------------------------
```

```
        ListaInt() ;
        ListaInt(const ListaInt& o) ;
        ListaInt& operator = (const ListaInt& o) ;
        //------------------------------
        bool llena() const ;
        int size() const ;
        void clear() ;
        //------------------------------
        void insertar(int pos, int dato) ;
            // PRECOND: ( ! llena() && pos >= 0 && pos <= size())
        void eliminar(int pos) ;
            // PRECOND: (pos >= 0 && pos < size())
        //------------------------------
        int acceder(int pos) const ;
            // PRECOND: (pos >= 0 && pos < size())
        void modificar(int pos, int dato);
            // PRECOND: (pos >= 0 && pos < size())
        //----------------------------------------------------------
    private:
        //----------------------------------------------------------
        //-- Ctes y Tipos Privados ---------------------------------
        //----------------------------------------------------------
        static const int MAX = 100;
        typedef std::array<int, MAX> Datos;
        //----------------------------------------------------------
        //-- Metodos Privados --------------------------------------
        //----------------------------------------------------------
        void abrir_hueco(int pos) ;
        void cerrar_hueco(int pos) ;
        //----------------------------------------------------------
        //-- Atributos Privados ------------------------------------
        //----------------------------------------------------------
        int sz;         // numero de elementos de la lista
        Datos v;        // contiene los elementos de la lista
        //----------------------------------------------------------
    };
}
#endif
//- fin: lista.hpp ---------------------------------------------
```

Implementación

```
//- fichero: lista.cpp -----------------------------------------
#include "lista.hpp"
#include <cassert>
namespace umalcc {
    //----------------------------------------------------------
    //-- Métodos Públicos --------------------------------------
    //----------------------------------------------------------
    // ListaInt::~ListaInt() { }               // Destructor Automático
    //---------------------------------
    ListaInt::ListaInt() : sz(0), v() { }       // Constructor por Defecto
    //---------------------------------
    ListaInt::ListaInt(const ListaInt& o)       // Constructor de Copia
        : sz(o.sz), v()
    {
        for (int i = 0; i < sz; ++i) {
            v[i] = o.v[i] ;
        }
```

```
}
//----------------------------------
ListaInt& ListaInt::operator = (const ListaInt& o)  // Op. de Asignación
{
    if (this != &o) {
        sz = o.sz ;
        for (int i = 0; i < sz; ++i) {
            v[i] = o.v[i] ;
        }
    }
    return *this ;
}
//----------------------------------
bool ListaInt::llena() const
{
    return sz == int(v.size());
}
//----------------------------------
int ListaInt::size() const
{
    return sz ;
}
//----------------------------------
void ListaInt::clear()
{
    sz = 0 ;
}
//----------------------------------
void ListaInt::insertar(int pos, int dato)
{
    assert( ! llena() && pos >= 0 && pos <= size()) ;
    abrir_hueco(pos) ;
    v[pos] = dato ;
}
//----------------------------------
void ListaInt::eliminar(int pos)
{
    assert(pos >= 0 && pos < size()) ;
    cerrar_hueco(pos) ;
}
//----------------------------------
int ListaInt::acceder(int pos) const
{
    assert(pos >= 0 && pos < size()) ;
    return v[pos] ;
}
//----------------------------------
void ListaInt::modificar(int pos, int dato)
{
    assert(pos >= 0 && pos < size()) ;
    v[pos] = dato;
}
//---------------------------------------------------------
//-- Metodos Privados -------------------------------------
//---------------------------------------------------------
void ListaInt::abrir_hueco(int pos)
{
    assert(sz < int(v.size())) ;
    for (int i = sz; i > pos; --i) {
```

```
            v[i] = v[i-1];
        }
        ++sz;                               // Ahora hay un elemento más
    }
    //----------------------------------
    void ListaInt::cerrar_hueco(int pos)
    {
        assert(sz > 0) ;
        --sz;                               // Ahora hay un elemento menos
        for (int i = pos; i < sz; ++i) {
            v[i] = v[i+1];
        }
    }
    //----------------------------------
}
//- fin: lista.cpp ---------------------------------------------
```

Utilización

```
//- fichero: main.cpp ----------------------------------------------
#include <iostream>
#include <cctype>
#include <cassert>
#include "lista.hpp"
using namespace std ;
using namespace umalcc ;
//------------------------------------------------------------------
void leer_pos(int& pos, int limite)
{
    assert(limite > 0);
    do {
        cout << "Introduzca posicion ( < " << limite << " ): " ;
        cin >> pos;
    } while (pos < 0 || pos >= limite);
}
//---------------------------------
void leer_dato(int& dato)
{
    cout << "Introduzca un dato: " ;
    cin >> dato;
}
//---------------------------------
void leer(ListaInt& lista)
{
    int dato ;
    lista.clear() ;
    cout << "Introduzca datos (0 -> FIN): " << endl ;
    cin >> dato ;
    while ((dato != 0)&&( ! lista.llena())) {
        lista.insertar(lista.size(), dato) ;
        cin >> dato ;
    }
}
//---------------------------------
void escribir(const ListaInt& lista)
{
    cout << "Lista: " ;
    for (int i = 0 ; i < lista.size() ; ++i) {
        cout << lista.acceder(i) << " " ;
```

```
    }
    cout << endl ;
}
//--------------------------------
void prueba_asg(const ListaInt& lista)
{
    cout << "Constructor de Copia" << endl ;
    ListaInt lst(lista) ;
    escribir(lst) ;
    cout << "Operador de Asignacion" << endl ;
    lst = lista ;
    escribir(lst) ;
}
//-------------------------------------------------------------------------
char menu()
{
    char op ;
    cout << endl ;
    cout << "X. Fin" << endl ;
    cout << "A. Leer Lista" << endl ;
    cout << "B. Borrar Lista" << endl ;
    cout << "C. Insertar Posicion" << endl ;
    cout << "D. Eliminar Posicion" << endl ;
    cout << "E. Acceder Posicion" << endl ;
    cout << "F. Modificar Posicion" << endl ;
    cout << "G. Prueba Copia y Asignacion" << endl ;
    do {
        cout << endl << "    Opcion: " ;
        cin >> op ;
        op = char(toupper(op)) ;
    } while (!((op == 'X')||((op >= 'A')&&(op <= 'G')))) ;
    cout << endl ;
    return op ;
}
//-------------------------------------------------------------------------
int main()
{
    ListaInt lista ;
    int dato ;
    int pos ;
    char op = ' ' ;
    do {
        op = menu() ;
        switch (op) {
        case 'A':
            leer(lista) ;
            escribir(lista) ;
            break ;
        case 'B':
            lista.clear() ;
            escribir(lista) ;
            break ;
        case 'C':
            if (lista.llena()) {
                cout << "Error: Lista llena" << endl ;
            } else {
                leer_pos(pos, lista.size()+1) ;
                leer_dato(dato) ;
                lista.insertar(pos, dato) ;
```

```
                escribir(lista) ;
            }
            break ;
        case 'D':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                lista.eliminar(pos) ;
                escribir(lista) ;
            }
            break ;
        case 'E':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                cout << "Lista[" << pos << "]: " << lista.acceder(pos) << endl ;
                escribir(lista) ;
            }
            break ;
        case 'F':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                leer_dato(dato) ;
                lista.modificar(pos, dato) ;
                escribir(lista) ;
            }
            break ;
        case 'G':
            prueba_asg(lista) ;
            break ;
        }
    } while (op != 'X') ;
}
//- fin: main.cpp ---------------------------------------------------
```

Capítulo 12

Memoria Dinámica. Punteros

Hasta ahora, todos los programas que se han visto en capítulos anteriores almacenan su estado interno por medio de variables que son automáticamente gestionadas por el compilador. Las variables son *creadas* cuando el flujo de ejecución entra en el ámbito de su definición (se reserva espacio en memoria y se crea el valor de su estado inicial), posteriormente se *manipula* el estado de la variable (accediendo o modificando su valor almacenado), y finalmente se *destruye* la variable cuando el flujo de ejecución sale del ámbito donde fue declarada la variable (liberando los recursos asociados a ella y la zona de memoria utilizada). A este tipo de variables gestionadas automáticamente por el compilador se las suele denominar *variables automáticas* (también variables locales), y residen en una zona de memoria gestionada automáticamente por el compilador, la *pila* de ejecución, donde se alojan y desalojan las variables locales (automáticas) pertenecientes al ámbito de ejecución de cada subprograma.

Así, el tiempo de vida de una determinada variable está condicionado por el ámbito de su declaración. Además, el número de variables automáticas utilizadas en un determinado programa está especificado explícitamente en el propio programa, y por lo tanto su capacidad de almacenamiento está también especificada y predeterminada por lo especificado explícitamente en el programa. Es decir, con la utilización única de variables automáticas, la capacidad de almacenamiento de un determinado programa está predeterminada desde el momento de su programación (tiempo de compilación), y no puede adaptarse a las necesidades reales de almacenamiento surgidas durante la ejecución del programa (tiempo de ejecución).[1]

La gestión de *memoria dinámica* surge como un mecanismo para que el propio programa, durante su ejecución (tiempo de ejecución), pueda solicitar (alojar) y liberar (desalojar) memoria según las necesidades surgidas durante una determinada ejecución, dependiendo de las circunstancias reales de cada momento de la ejecución del programa en un determinado entorno. Esta ventaja adicional viene acompañada por un determinado coste asociado a la mayor complejidad que requiere su gestión, ya que en el caso de las variables automáticas, es el propio compilador el encargado de su gestión, sin embargo en el caso de las *variables dinámicas* es el propio programador el que debe, mediante código software, gestionar el tiempo de vida de cada variable dinámica, cuando debe ser alojada y creada, como será utilizada, y finalmente cuando debe ser destruida y desalojada. Adicionalmente, como parte de esta gestión de la memoria dinámica por el propio programador, la memoria dinámica pasa a ser un *recurso* que debe gestionar el programador, y se debe preocupar de su alojo y de su liberación, poniendo especial cuidado y énfasis en no perder recursos (perder zonas de memoria sin liberar y sin capacidad de acceso).

[1] En realidad esto no es completamente cierto, ya que en el caso de subprogramas recursivos, cada invocación recursiva en tiempo de ejecución tiene la capacidad de alojar nuevas variables que serán posteriormente desalojadas automáticamente cuando la llamada recursiva finaliza.

12.1. Punteros

El *tipo puntero* es un tipo *simple* que permite a un determinado programa acceder a posiciones concretas de memoria, y más específicamente a determinadas zonas de la memoria dinámica. Aunque el lenguaje de programación C++ permite otras utilizaciones más diversas del tipo puntero, en este capítulo sólo se utilizará el tipo puntero para acceder a zonas de memoria dinámica.

Así, una determinada variable de tipo puntero apunta (o referencia) a una determinada entidad (variable) de un determinado tipo alojada en la zona de memoria dinámica. Por lo tanto, para un determinado tipo puntero, se debe especificar también el tipo de la variable (en memoria dinámica) a la que apunta, el cual define el espacio que ocupa en memoria y las operaciones (y métodos) que se le pueden aplicar, entre otras cosas.

De este modo, cuando un programa gestiona la memoria dinámica a través de punteros, debe manejar y gestionar por una parte la propia variable de tipo puntero, y por otra parte la variable dinámica apuntada por éste.

Un tipo puntero se define utilizando la palabra reservada `typedef` seguida del tipo de la variable dinámica apuntada, un asterisco para indicar que es un **puntero** a una variable de dicho tipo, y el identificador que denomina al tipo. Por ejemplo:

```
typedef int* PInt ;          // Tipo Puntero a Entero

struct Persona {             // Tipo Persona
    string nombre ;
    string telefono ;
    int edad ;
} ;
typedef Persona* PPersona ; // Tipo Puntero a Persona
```

Así, el tipo `PInt` es el tipo de una variable que apunta a una variable dinámica de tipo `int`. Del mismo modo, el tipo `PPersona` es el tipo de una variable que apunta a una variable dinámica de tipo `Persona`.

Es importante remarcar que el *tipo puntero*, en sí mismo, es un **tipo simple**, aunque el tipo apuntado puede ser tanto un tipo simple, como un tipo compuesto.

Es posible definir variables de los tipos especificados anteriormente. Nótese que estas variables (`p1` y `p2` en el siguiente ejemplo) son variables automáticas (gestionadas automáticamente por el compilador), es decir, se crean automáticamente (con un valor indeterminado) al entrar el flujo de ejecución en el ámbito de visibilidad de la variable, y posteriormente se destruyen automáticamente cuando el flujo de ejecución sale del ámbito de visibilidad de la variable. Por otra parte, las variables apuntadas por ellos son variables dinámicas (gestionadas por el programador), es decir el programador se encargará de solicitar la memoria dinámica cuando sea necesaria y de liberarla cuando ya no sea necesaria, durante la ejecución del programa. En el siguiente ejemplo, si las variables se definen sin inicializar, entonces tendrán un valor inicial inespecificado:

```
int main()
{
    PInt p1 ;
    PPersona p2 ;
}
```

p1: ?
p2: ?

La constante `NULL` es una constante especial de tipo puntero que indica que una determinada variable de tipo puntero no apunta a nada, es decir, especifica que la variable de tipo puntero que contenga el valor `NULL` no apunta a ninguna zona de la memoria dinámica. Para utilizar la constante `NULL` se debe incluir la biblioteca estándar `<cstddef>`. Así, se pueden definir las variables `p1` y `p2` e inicializarlas a un valor indicando que no apuntan a nada.

```
#include <cstddef>
int main()
{
    PInt p1 = NULL ;
    PPersona p2 = NULL ;
}
```

p1: /
p2: /

12.2. Gestión de Memoria Dinámica

La memoria dinámica la debe gestionar el propio programador, por lo que cuando necesite crear una determinada variable dinámica, debe *solicitar memoria dinámica* con el operador `new` seguido por el tipo de la variable dinámica a crear. Este operador (`new`) realiza dos acciones principales, primero **aloja** (reserva) espacio en memoria dinámica para albergar a la variable, y después **crea** (invocando al constructor especificado) el contenido de la variable dinámica. Finalmente, a la variable `ptr` se le asigna el valor del puntero (una dirección de memoria) que apunta a la variable dinámica creada por el operador `new`. Por ejemplo, para crear una variable dinámica del tipo `Persona` definido anteriormente utilizando el constructor por defecto de dicho tipo.

```
int main()
{
    PPersona ptr ;
    ptr = new Persona ;
}
```

ptr: ???

En caso de que el tipo de la variable dinámica tenga otros constructores definidos, es posible utilizarlos en la construcción del objeto en memoria dinámica. Por ejemplo, suponiendo que el tipo `Persona` tuviese un constructor que reciba el nombre, teléfono y edad de la persona:

```
int main()
{
    PPersona ptr ;
    ptr = new Persona("pepe", "111", 5) ;
}
```

ptr: pepe 111 5

Posteriormente, tras manipular adecuadamente, según las características del programa, la memoria dinámica alojada, llegará un momento en que dicha variable dinámica ya no sea necesaria, y su tiempo de vida llegue a su fin. En este caso, el programador debe *liberar* explícitamente dicha variable dinámica mediante el operador `delete` de la siguiente forma:

```
int main()
{
    PPersona ptr ;      // Creación automática de la variable PTR
    ptr = new Persona ; // Creación de la variable dinámica anónima

    // manipulación ...

    delete ptr ;        // Destrucción de la variable dinámica anónima
}                       // Destrucción automática de la variable PTR
```

ptr: pepe 111 5

liberada

ptr: ?

La sentencia `delete ptr` realiza dos acciones principales, primero **destruye** la variable dinámica (invocando a su destructor), y después **desaloja** (libera) la memoria dinámica reservada para dicha variable. Finalmente la variable local `ptr` queda con un valor inespecificado, y será destruida automáticamente por el compilador cuando el flujo de ejecución salga de su ámbito de declaración.

Si se ejecuta la operación `delete` sobre una variable de tipo puntero que tiene el valor `NULL`, entonces esta operación no hace nada.

En caso de que no se libere (mediante el operador `delete`) la memoria dinámica apuntada por la variable `ptr`, y esta variable sea destruida al terminar su tiempo de vida (su ámbito de visibilidad), entonces se *perderá* la memoria dinámica a la que apunta, con la consiguiente **pérdida de recursos** que ello conlleva.

```
int main()
{
    PPersona ptr ;
    ptr = new Persona("pepe", "111", 5) ;

    // manipulación ...

    // no se libera la memoria dinámica apuntada por ptr
    // se destruye la variable local ptr
}
```

ptr: pepe 111 5

perdida

pepe 111 5

12.3. Operaciones con Variables de Tipo Puntero

Asignación de Variables de Tipo Puntero

El puntero nulo (`NULL`) se puede asignar a cualquier variable de tipo puntero. Por ejemplo:

```
int main()
{
    PPersona p1 ;
    // ...
    p1 = NULL ;
    // ...
}
```

El resultado de crear una variable dinámica con el operador `new` se puede asignar a una variable de tipo puntero al tipo de la variable dinámica creada. Por ejemplo:

```
int main()
{
    PPersona p1 ;
    // ...
    p1 = new Persona("pepe", "111", 5) ;
    // ...
}
```

Así mismo, a una variable de tipo puntero se le puede asignar el valor de otra variable puntero. En este caso, ambas variables de tipo puntero apuntarán a la misma variable dinámica, que será compartida por ambas. Si se libera la variable dinámica apuntada por una de ellas, la variable dinámica compartida se destruye, su memoria se desaloja y ambas variables locales de tipo puntero quedan con un valor inespecificado.

```
int main()
{
    PPersona p1 = new Persona("pepe", "111", 5) ;
    PPersona p2 ;
    // ...
    p2 = p1 ;
    // ...
    delete p1 ;
}
```

En la operación de asignación, el valor anterior que tuviese la variable de tipo puntero se pierde, por lo que habrá que tener especial cuidado de que no se pierda la variable dinámica que tuviese asignada, si tuviese alguna.

```
int main()
{
    PPersona p1 = new Persona("pepe", "111", 5) ;
    // ...
    p1 = NULL ; // se pierde el valor anterior
    // ...
    delete p1 ;
}
```

Desreferenciación de una Variable de Tipo Puntero

Para acceder a una variable dinámica apuntada por una variable de tipo puntero, se utiliza el operador unario *asterisco* (`*`) precediendo al nombre de la variable de tipo puntero a través de la cual es apuntada. Por ejemplo, si `ptr` es una variable local de tipo puntero que apunta a una variable dinámica de tipo `Persona`, entonces `*ptr` es la variable dinámica apuntada, y se trata de igual forma que cualquier otra variable de tipo `Persona`.

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5) ;

    Persona p = *ptr ;  // Asigna el contenido de la variable dinámica a la variable p

    *ptr = p ;          // Asigna el contenido de la variable p a la variable dinámica

    delete ptr ;        // destruye la variable dinámica y libera su espacio de memoria
}
```

Sin embargo, si una variable de tipo puntero tiene el valor `NULL`, entonces *desreferenciar* la variable produce un **error** en tiempo de ejecución que aborta la ejecución del programa. Así mismo, *desreferenciar* un puntero con valor inespecificado produce un comportamiento **anómalo** en tiempo de ejecución.

Es posible, así mismo, acceder a los elementos de la variable apuntada mediante el operador de desreferenciación. Por ejemplo:

```
int main()
{
    PPersona ptr = new Persona ;

    (*ptr).nombre = "pepe" ;
    (*ptr).telefono = "111" ;
    (*ptr).edad = 5 ;

    delete ptr ;
}
```

Nótese que el uso de los paréntesis es obligatorio debido a que el operador punto (`.`) tiene mayor precedencia que el operador de desreferenciación (`*`). Por ello, en el caso de acceder a los campos de un registro en memoria dinámica a través de una variable de tipo puntero, es más adecuado utilizar el operador de desreferenciación (`->`). Por ejemplo:

```
int main()
{
    PPersona ptr = new Persona ;

    ptr->nombre = "pepe" ;
    ptr->telefono = "111" ;
    ptr->edad = 5 ;

    delete ptr ;
}
```

Este operador también se utiliza para invocar a métodos de un objeto si éste se encuentra alojado en memoria dinámica. Por ejemplo:

```
#include <iostream>
using namespace std ;
class Numero {
public:
    Numero(int v) : val(v) {}
    int valor() const { return val ; }
private:
    int val ;
} ;
typedef Numero* PNumero ;
int main()
{
```

```
    PNumero ptr = new Numero(5) ;

    cout << ptr->valor() << endl ;

    delete ptr ;
}
```

Comparación de Variables de Tipo Puntero

Las variables del mismo tipo puntero se pueden comparar entre ellas por igualdad (`==`) o desigualdad (`!=`), para comprobar si apuntan a la misma variable dinámica. Así mismo, también se pueden comparar por igualdad o desigualdad con el puntero nulo (`NULL`) para saber si apunta a alguna variable dinámica, o por el contrario no apunta a nada. Por ejemplo:

```
int main()
{
    PPersona p1, p2 ;
    // ...
    if (p1 == p2) {
        // ...
    }
    if (p1 != NULL) {
        // ...
    }
}
```

12.4. Paso de Parámetros de Variables de Tipo Puntero

El tipo puntero es un *tipo simple*, y por lo tanto se tratará como tal. En caso de paso de parámetros de tipo puntero, si es un parámetro de entrada, entonces se utilizará el *paso por valor*, y si es un parámetro de salida o de entrada/salida, entonces se utilizará el *paso por referencia*.

Hay que ser consciente de que un parámetro de tipo puntero puede apuntar a una variable dinámica, y en este caso, a partir del parámetro se puede acceder a la variable apuntada.

Así, si el parámetro se pasa *por valor*, entonces se copia el valor del puntero del parámetro actual (en la invocación) al parámetro formal (en el subprograma), por lo que ambos apuntarán a la misma variable dinámica compartida, y en este caso, si se modifica el valor almacenado en la variable dinámica, este valor se verá afectado, así mismo, en el exterior del subprograma, aunque el parámetro haya sido pasado por valor.

Por otra parte, las funciones también pueden devolver valores de tipo puntero.

```
void modificar(PPersona& p) ;

PPersona buscar(PPersona l, const string& nombre) ;
```

12.5. Listas Enlazadas Lineales

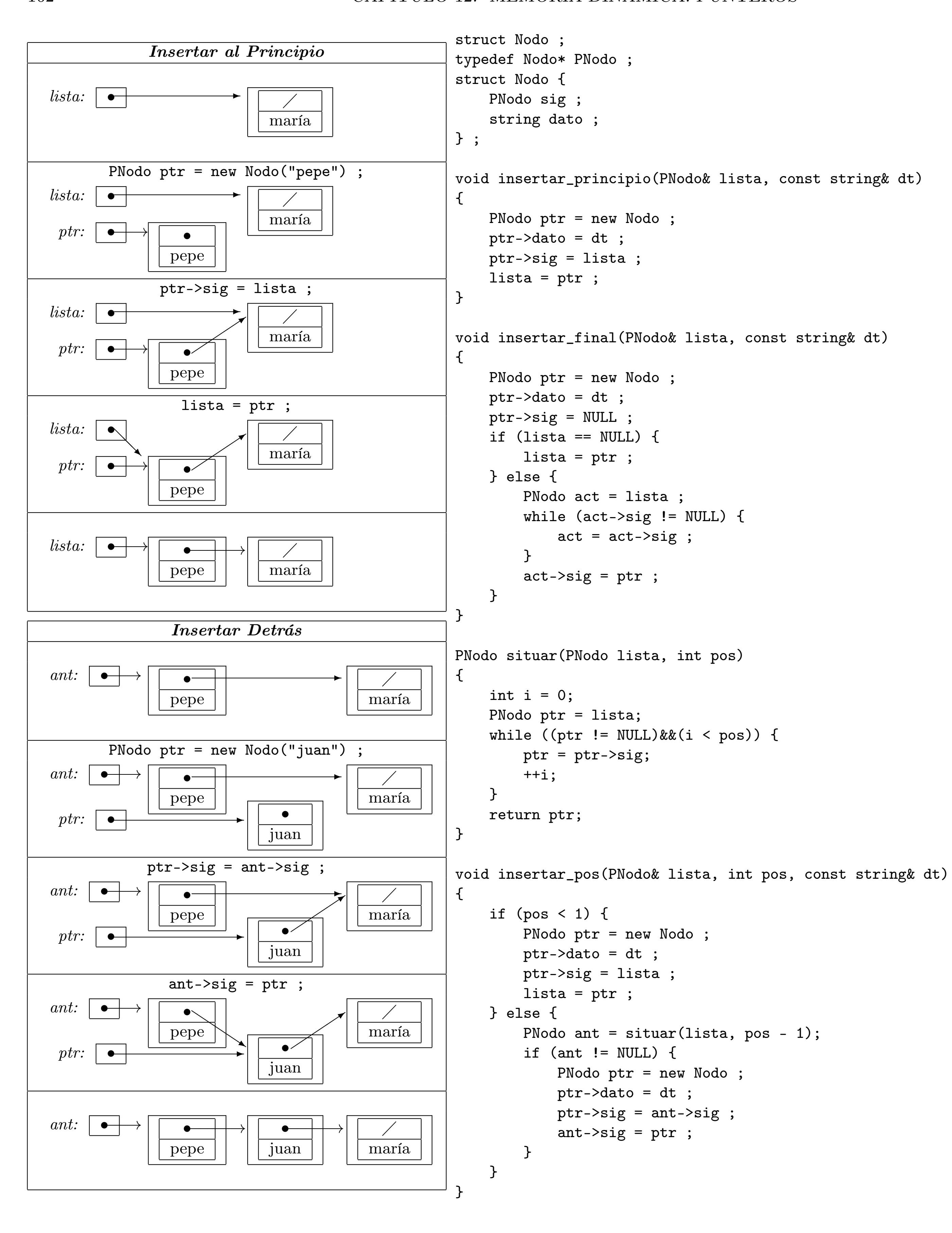
Una de las principales aplicaciones de la Memoria Dinámica es el uso de estructuras enlazadas, de tal forma que un campo o atributo de la variable dinámica es a su vez también de tipo puntero, por lo que puede apuntar a otra variable dinámica que también tenga un campo o atributo de tipo puntero, el cual puede volver a apuntar a otra variable dinámica, y así sucesivamente, tantas veces como sea necesario, hasta que un puntero con el valor `NULL` indique el final de la estructura enlazada (lista enlazada).

Así, en este caso, vemos que un campo de la estructura es de tipo puntero a la propia estructura, por lo que es necesario definir el tipo puntero antes de definir la estructura. Sin embargo, la estructura todavía no ha sido definida, por lo que no se puede definir un puntero a ella. Por ello es necesario realizar una *declaración adelantada* de un *tipo incompleto* del tipo de la variable

dinámica, donde se declara que un determinado identificador es una estructura o clase, pero no se definen sus componentes.

```
struct Nodo ;          // Declaración adelantada del tipo incompleto Nodo
typedef Nodo* PNodo ; // Definición de tipo Puntero a tipo incompleto Nodo
struct Nodo {          // Definición del tipo Nodo
    PNodo sig ;        // Enlace a la siguiente estructura dinámica
    string dato ;      // Dato almacenado en la lista
} ;
void escribir(PNodo lista)
{
    PNodo ptr = lista;
    while (ptr != NULL) {
        cout << ptr->dato << endl ;
        ptr = ptr->sig ;
    }
}
PNodo buscar(PNodo lista, const string& dt)
{
    PNodo ptr = lista ;
    while ((ptr != NULL)&&(ptr->dato != dt)) {
        ptr = ptr->sig ;
    }
    return ptr ;
}
PNodo leer_inversa()
{
    PNodo lista = NULL ;
    string dt ;
    cin >> dt ;
    while (dt != "fin") {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lista ;
        lista = ptr ;
        cin >> dt ;
    }
    return lista ;
}
void destruir(PNodo& lista)
{
    while (lista != NULL) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
}
int main()
{
    PNodo lista ;
    lista = leer_inversa() ;
    escribir(lista) ;
    PNodo ptr = buscar(lista, "juan");
    if (ptr != NULL) {
        cout << ptr->dato << endl;
    }
    destruir(lista) ;
}
```

lista: pepe juan maría



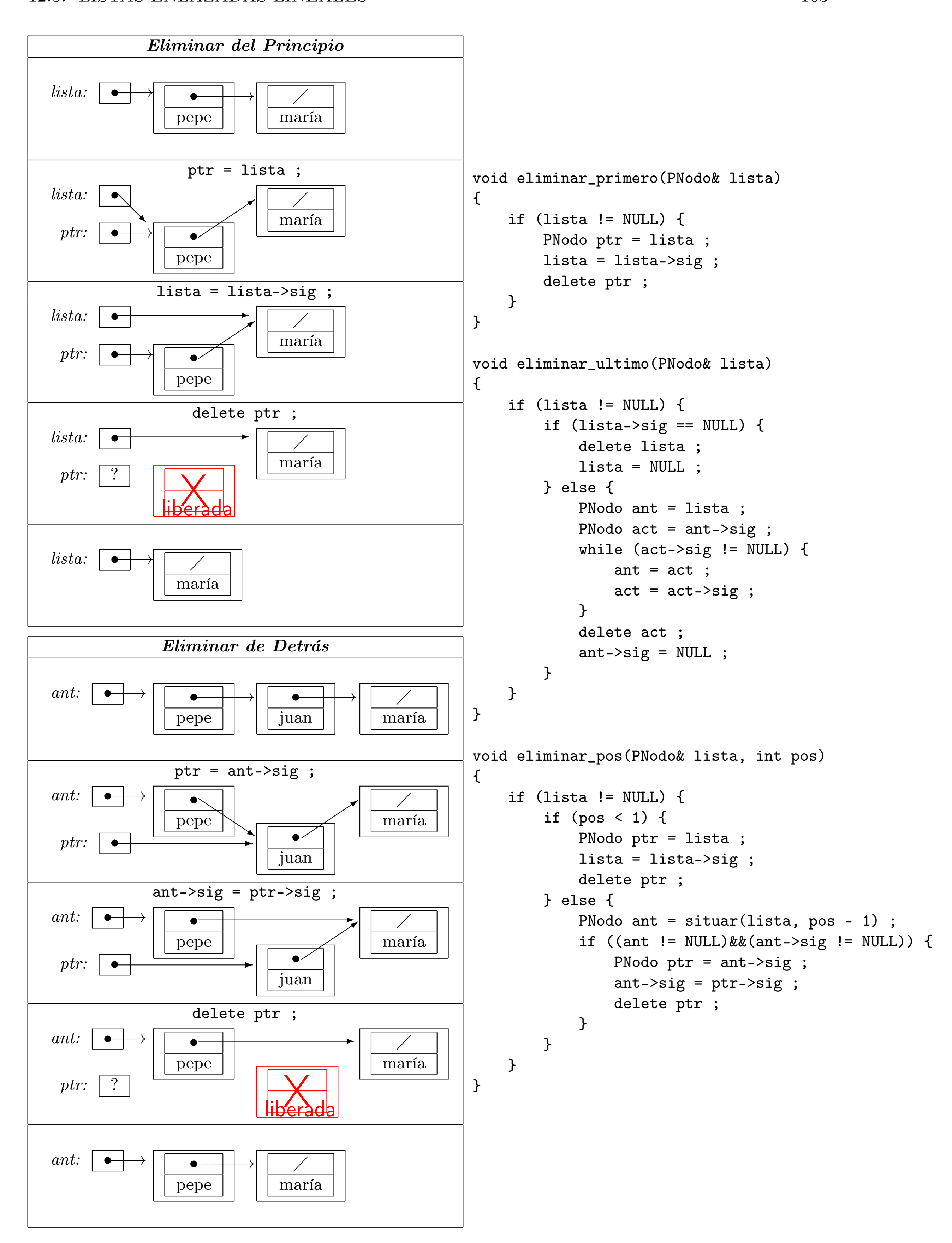

```
struct Nodo ;
typedef Nodo* PNodo ;
struct Nodo {
    PNodo sig ;
    string dato ;
} ;

void insertar_principio(PNodo& lista, const string& dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    ptr->sig = lista ;
    lista = ptr ;
}

void insertar_final(PNodo& lista, const string& dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    ptr->sig = NULL ;
    if (lista == NULL) {
        lista = ptr ;
    } else {
        PNodo act = lista ;
        while (act->sig != NULL) {
            act = act->sig ;
        }
        act->sig = ptr ;
    }
}

PNodo situar(PNodo lista, int pos)
{
    int i = 0;
    PNodo ptr = lista;
    while ((ptr != NULL)&&(i < pos)) {
        ptr = ptr->sig;
        ++i;
    }
    return ptr;
}

void insertar_pos(PNodo& lista, int pos, const string& dt)
{
    if (pos < 1) {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lista ;
        lista = ptr ;
    } else {
        PNodo ant = situar(lista, pos - 1);
        if (ant != NULL) {
            PNodo ptr = new Nodo ;
            ptr->dato = dt ;
            ptr->sig = ant->sig ;
            ant->sig = ptr ;
        }
    }
}
```




```
void eliminar_primero(PNodo& lista)
{
    if (lista != NULL) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
}

void eliminar_ultimo(PNodo& lista)
{
    if (lista != NULL) {
        if (lista->sig == NULL) {
            delete lista ;
            lista = NULL ;
        } else {
            PNodo ant = lista ;
            PNodo act = ant->sig ;
            while (act->sig != NULL) {
                ant = act ;
                act = act->sig ;
            }
            delete act ;
            ant->sig = NULL ;
        }
    }
}

void eliminar_pos(PNodo& lista, int pos)
{
    if (lista != NULL) {
        if (pos < 1) {
            PNodo ptr = lista ;
            lista = lista->sig ;
            delete ptr ;
        } else {
            PNodo ant = situar(lista, pos - 1) ;
            if ((ant != NULL)&&(ant->sig != NULL)) {
                PNodo ptr = ant->sig ;
                ant->sig = ptr->sig ;
                delete ptr ;
            }
        }
    }
}
```

```
void insertar_ord(PNodo& lista,
                  const string& dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    if ((lista==NULL)||(dt < lista->dato)) {
        ptr->sig = lista ;
        lista = ptr ;
    } else {
        PNodo ant = lista ;
        PNodo act = ant->sig ;
        while ((act!=NULL)&&(act->dato<=dt)){
            ant = act ;
            act = act->sig ;
        }
        ptr->sig = ant->sig ;
        ant->sig = ptr ;
    }
}
```

```
void eliminar_elem(PNodo& lista, const string& dt, bool& ok)
{
    ok = false ;
    if (lista != NULL) {
        if (lista->dato == dt) {
            PNodo ptr = lista ;
            lista = lista->sig ;
            delete ptr ;
            ok = true ;
        } else {
            PNodo ant = lista ;
            PNodo act = ant->sig ;
            while ((act != NULL)&&(act->dato != dt)) {
                ant = act ;
                act = act->sig ;
            }
            if (act != NULL) {
                ant->sig = act->sig ;
                delete act ;
                ok = true ;
            }
        }
    }
}
```

```
PNodo duplicar(PNodo lista)
{
    PNodo nueva = NULL;
    if (lista != NULL) {
        nueva = new Nodo ;
        nueva->dato = lista->dato ;
        PNodo u = nueva ;
        PNodo p = lista->sig ;
        while (p != NULL) {
            u->sig = new Nodo ;
            u->sig->dato = p->dato ;
            u = u->sig ;
            p = p->sig ;
        }
        u->sig = NULL ;
    }
    return nueva;
}
```

```
void purgar(PNodo& lista, const string& dt)
{
    while ((lista != NULL)&&(dt == lista->dato)) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
    if (lista != NULL) {
        PNodo ant = lista;
        PNodo act = lista->sig;
        while (act != NULL) {
            if (dt == act->dato) {
                ant->sig = act->sig ;
                delete act ;
            } else {
                ant = act;
            }
            act = ant->sig;
        }
    }
}
```

```
void insertar_ultimo(PNodo& lista,
                     PNodo& ult,
                     const string& dt)
{
    PNodo p = new Nodo ;
    p->dato = dt ;
    p->sig = NULL ;
    if (lista == NULL) {
        lista = p ;
    } else {
        ult->sig = p ;
    }
    ult = p ;
}
```

```
PNodo leer()
{
    PNodo lista = NULL ;
    PNodo ult = NULL ;
    string dt ;
    cin >> dt ;
    while (dt != "fin") {
        insertar_ultimo(lista, ult, dt) ;
        cin >> dt ;
    }
    return lista ;
}
```

12.6. Abstracción en la Gestión de Memoria Dinámica

La gestión de memoria dinámica por parte del programador se basa en estructuras de programación de *bajo nivel*, las cuales son propensas a errores de programación y pérdida de recursos de memoria. Además, entremezclar sentencias de gestión de memoria, de bajo nivel, con sentencias aplicadas al dominio de problema a resolver suele dar lugar a código no legible y propenso a errores.

Por lo tanto se hace necesario aplicar niveles de abstracción que aislen la gestión de memoria dinámica (de bajo nivel) del resto del código más directamente relacionado con la solución del problema. Para ello, los tipos abstractos de datos proporcionan el mecanismo adecuado para aplicar la abstracción a estas estructuras de datos basadas en la gestión de memoria dinámica, además de proporcionar una herramienta adecuada para la gestión de memoria dinámica, ya que los destructores se pueden encargar de liberar los recursos asociados a un determinado objeto.

Así, vemos que será necesario duplicar la información almacenada en la memoria dinámica al copiar y asignar objetos, así como también será necesario liberar la memoria dinámica antes de asignar nueva información o de destruir el objeto.

- El *Destructor* debe liberar la memoria dinámica antes de destruir el objeto.
- El *Ctor-Defecto* debe inicializar adecuadamente los atributos de tipo puntero.
- El *Ctor-Copia* debe duplicar la memoria dinámica del objeto a copiar.
- El *Op-Asignación* debe liberar la memoria dinámica actual y duplicar la memoria dinámica del objeto a asignar.
- Los *métodos* de la clase permiten manipular la estructura de datos, proporcionando abstracción sobre su complejidad y representación interna.
- El acceso restringido a la representación interna impide una manipulación externa propensa a errores.

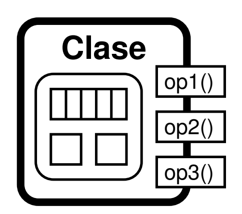
```
class Lista {
public:
    ~Lista() { destruir(lista) ; }
    Lista() : sz(0), lista(NULL) { }
    Lista(const Lista& o)
        : sz(o.sz), lista(duplicar(o.lista)) { }
    Lista& operator = (const Lista& o)
    {
        if (this != &o) {
            destruir(lista) ;
            sz = o.sz ;
            lista = duplicar(o.lista) ;
        }
        return *this ;
    }
    void insertar(int pos, int d) { ... }
    void eliminar(int pos) { ... }
    // ...
private:
    struct Nodo ;
    typedef Nodo* PNodo ;
    struct Nodo {
        PNodo sig ;
        int dato ;
    } ;
    //-- Métodos privados --
    void destruir(PNodo& l) const { ... }
    PNodo duplicar(PNodo l) const { ... }
```

```
    // ...
    //-- Atributos privados --
    int sz ;
    PNodo lista ;
} ;
```

Capítulo 13

Introducción a la Programación Orientada a Objetos

La *Programación Orientada a Objetos* extiende los conceptos fundamentales de *abstracción* y *encapsulación* de los Tipos Abstractos de Datos (TADs), añadiendo los conceptos de *Herencia*, *Polimorfismo*, y *Vinculación Dinámica*.



Se mantiene el concepto de *Objetos* como instancias de *Clases*, los cuales son entidades activas que encapsulan datos y algoritmos, donde los *Atributos* contienen el estado y la representación interna del objeto, cuyo acceso está restringido, y los *Métodos* permiten la manipulación e interacción entre objetos. Así, una *Clase* define una *abstracción*, y los *métodos* definen su comportamiento. Así, las características de un determinado *objeto*, su estado y su comportamiento, están determinadas por la *clase* a la que pertenece. Del mismo modo, el *objeto* podrá ser manipulado e interactuar con otros objetos a través de los *métodos* definidos por la *Clase* a la que pertenece.

La *Programación Orientada a Objetos* proporciona un mecanismo adecuado para el diseño y desarrollo de software complejo, modular, reusable, adaptable y extensible.

13.0.1. Métodos y Atributos

Los *métodos* son algoritmos *especiales* definidos por la *clase*, y se aplican sobre los objetos, manipulando el estado interno del objeto sobre el que se aplican.

La invocación a métodos puede llevar parámetros asociados, así como producir un resultado, además de manipular el estado interno del objeto sobre el que se aplica. Para invocar a un determinado método sobre un objeto, ese método debe estar definido por la clase a la que el objeto pertenece.

Los *atributos* estan protegidos, de tal forma que sólo se permite su acceso y manipulación a través de los métodos definidos por la clase. Los *atributos* almacenan los valores del estado interno de cada objeto, considerando que cada objeto tiene un estado interno asociado, independiente de los otros objetos.

13.1. Herencia, Polimorfismo y Vinculación Dinámica

La *Herencia* permite modelar relaciones *"es un"* entre clases, definiendo *jerarquías* de clases. Permite definir una nueva *clase derivada* (o sub-clase) como una especialización o extensión de una *clase base* (o super-clase) más general, donde la clase derivada *hereda* tanto los atributos, como los métodos definidos por la clase base (reusabilidad del código), y la clase derivada puede definir nuevos atributos y nuevos métodos (extensibilidad), así como también *redefinir* métodos de la clase base. Las relaciones de herencia pueden mostrarse adecuadamente utilizando los diagramas UML de clases.



El *polimorfismo* permite que un objeto de una *clase derivada* pueda ser considerado y utilizado como si fuera un objeto de la *clase base*, proporcionando un soporte adecuado para el *Principio de sustitución*, mediante el cual, un objeto de la clase derivada puede sustituir a un objeto de la clase base, allí donde sea necesario. Sin embargo, la dirección de correspondencia opuesta no se mantiene, ya que no todos los objetos de la clase base son también objetos de la clase derivada.

La *vinculación dinámica* permite que las clases derivadas puedan *redefinir* el comportamiento de los métodos definidos en la clase base. Así, en *contextos polimórficos*, gracias a la *vinculación dinámica*, los métodos invocados se seleccionan adecuadamente, en tiempo de ejecución, dependiendo del *tipo real* del objeto, y no del *tipo aparente*. Es decir, si se invoca a un determinado método sobre un objeto de una clase derivada que haya *redefinido* la implementacion de ese método, entonces se ejecutará el código del método redefinido en la clase derivada, incluso aunque la invocación se haya producido en un contexto polimórfico donde el objeto de la clase derivada haya sustituido a un objeto de la clase base.

13.2. Definición e Implementación de Clases Polimórficas

C++ proporciona un soporte adecuado tanto al paradigma de *Tipos Abstractos de Datos*, mediante *clases no-polimorficas* (véase 11), como al paradigma de *Programación Orientada a Objetos*, mediante *clases polimórficas*.

En C++, se puede definir una clase *derivada* de una clase *base* especificando el delimitador (`:`) después del identificador de la clase derivada, seguida de la palabra reservada `public` (para especificar herencia pública) y del identificador de la clase base de la que hereda.

```
class Base {
    // ...
};
class Derivada : public Base {
    // ...
};
```

Así mismo, también es posible definir un nuevo ámbito de visibilidad (`protected`) como un ámbito de acceso restringido que permite el acceso desde la propia clase, así como también desde las *clases derivadas*. De esta forma se proporciona un ambito adecuado donde definir métodos *protegidos* en la clase base para que las clases derivadas puedan manipular adecuadamente el estado interno de la clase base cuando sea necesario.

Los constructores de las clases derivadas pueden invocar *explícitamente*, al principio de la lista de inicialización, a los contructores de las clases base. En caso de que el constructor de la clase base no sea invocado *explícitamente* desde la lista de inicialización del constructor de la clase derivada,

entonces se añadirá automáticamente una invocación *implícita* al constructor por *defecto* de la clase base.

Con objeto de proporcionar un soporte adecuado a la vinculación dinámica en contextos polimórficos, tanto el *destructor* como todos los *métodos públicos* de cada clase polimórfica deben ser especificados con la palabra reservada `virtual` en la definición de la clase (en el fichero *hpp*), pero no en su implementación (en el fichero *cpp*).

Además, El *operador de asignación* no funciona adecuadamente en estos *contextos polimórficos*, por lo que se debe desactivar y sustituir por un método público virtual `clone()`. Para desactivar el operador de asignación, es suficiente con declarar su cabecera en la sección privada de la definición de la clase, dejando su suerpo sin implementar.

La implementación del método público virtual `clone()` debe devolver un puntero a un nuevo objeto de la clase creado como una *copia* del objeto actual, invocando para ello al *constructor de copia* de la clase, que debería estar definido dentro del ámbito de acceso `protected`, para evitar que pueda ser utilizado externamente, ya que es el metodo `clone()` el que debe ser invocado para crear una copia de un objeto polimórfico.

Por ejemplo, a continuación se muestra la definición e implementación de la clase polimórfica `Vehiculo`.

```
#ifndef vehiculo_hpp_
#define vehiculo_hpp_
#include <string>
namespace umalcc {
    class Vehiculo {
    public:
        Vehiculo(const std::string& i) ;
        virtual ~Vehiculo() ;
        virtual std::string id() const ;
        virtual void estacionar() ;
        virtual void mover() ;
        virtual Vehiculo* clone() const ;
    protected:
        Vehiculo(const Vehiculo& o) ;
        virtual void mover(int x) ;
    private:
        Vehiculo& operator=(const Vehiculo&) ;
        //---- Atributos ----
        std::string ident;
        int posicion ;
    };
}
#endif
```

```
#include "vehiculo.hpp"
namespace umalcc {
    Vehiculo::~Vehiculo() {}
    Vehiculo::Vehiculo(const std::string& i)
        : ident(i), posicion(0) {}
    Vehiculo::Vehiculo(const Vehiculo& o)
        : ident(o.ident), posicion(o.posicion) {}
    std::string Vehiculo::id() const {
        return ident;
    }
    void Vehiculo::estacionar() {
        posicion = 0 ;
    }
    void Vehiculo::mover() {
        ++posicion ;
    }
    void Vehiculo::mover(int x) {
        posicion += x ;
    }
    Vehiculo* Vehiculo::clone() const {
        return new Vehiculo(*this) ;
    }
}
```

Así, en la definición de la clase polimórfica `Vehiculo` del ejemplo anterior, podemos observar que tanto el *destructor* como *todos* los métodos públicos son *virtuales*. Esta definición es muy importante porque es el mecanismo que proporciona un soporte adecuado a la vinculación dinámica en contextos polimórficos.

Además, el operador de asignación ha sido desactivado, declarando su cabecera en la zona privada, y dejando su cuerpo sin implementación. El método público virtual `clone()` permite crear una copia del objeto actual, y funciona adecuadamente en contextos polimórficos, gracias a la vinculación dinámica. Nótese como se crea dinámicamente (en el *heap*) un nuevo objeto como copia del actual.

Hay un nuevo ámbito de acceso protegido, sólo accesible a la clase actual y a las clases derivadas, donde se definen aquellos métodos que permitirán a las clases derivadas manipular adecuadamente el estado interno de esta clase base. El constructor de copia se define en esta zona protegida, para permitir su invocación desde el método `clone()`, así como desde el constructor de copia de las clases derivadas. Pero sin embargo, se prohibe su invocación desde cualquier otro ámbito.

A continuación se muestra la definición e implementación de la clase polimórfica `Automovil`, como clase derivada de `Vehiculo`.

```
#ifndef automovil_hpp_
#define automovil_hpp_
#include "vehiculo.hpp"
#include <string>
namespace umalcc {
    class Automovil : public Vehiculo {
    public:
        Automovil(const std::string& i) ;
        virtual ~Automovil() ;
        virtual void mover() ;
        virtual void repostar(int litros) ;
        virtual Automovil* clone() const ;
    protected:
        Automovil(const Automovil& o) ;
    private:
        //---- Atributos ----
        int deposito ;
    };
}
#endif
```

```
#include "automovil.hpp"
namespace umalcc {
    Automovil::~Automovil() {}
    Automovil::Automovil(const std::string& i)
        : Vehiculo(i), deposito(0) {}
    Automovil::Automovil(const Automovil& o)
        : Vehiculo(o), deposito(o.deposito) {}
    void Automovil::mover() {
        if (deposito > 0) {
            Vehiculo::mover(50) ;
            --deposito ;
        }
    }
    void Automovil::repostar(int litros) {
        deposito += litros ;
    }
    Automovil* Automovil::clone() const {
        return new Automovil(*this) ;
    }
}
```

Así, la clase polimórfica `Automovil` hereda tanto los atributos (`ident` y `posicion`) como los métodos (`id()`, `estacionar()`, `mover()` y `clone()`) de la clase base `vehiculo`, pero *redefine* los métodos `mover()` y `clone()` de la clase base, proporcionando una nueva implementación y comportamiento. Además, añade un nuevo atributo (`deposito`) y un nuevo método (`repostar()`) a la clase `Automovil`.

En la definición de la clase derivada ya no es necesario desactivar el operador de asignación, ya que al estar desactivado en la clase base, entonces permanece desactivado para toda la jerarquía de clases que deriva de ella.

El destructor de la clase derivada ejecutará lo que se especifique en el cuerpo del mismo, y posteriormente invocará automáticamente al destructor de los atributos y al destructor de la clase base.

Se puede apreciar como tanto el constructor específico como el constructor de copia de la clase `Automovil` invocan en la lista de inicialización, en orden, tanto al constructor de la clase base, como al constructor de cada atributo.

En la implementación de un determinado método, se puede invocar directamente a los métodos de la clase base, tanto *públicos* como *protegidos*. Además, en caso de que se quiera invocar *específicamente* a un determinado método definido en una determinada clase base (sin que le afecte la vinculación dinámica), se puede hacer especificando el nombre de la clase donde se ha definido el método, seguido por el cualificador de ámbito (`::`) y del nombre del método, por ejemplo `Vehiculo::mover(50)`.

A continuación se muestra la definición e implementación de la clase polimórfica `Bicicleta`, como clase derivada de `Vehiculo`.

```
#ifndef bicicleta_hpp_
#define bicicleta_hpp_
#include "vehiculo.hpp"
#include <string>
namespace umalcc {
    class Bicicleta : public Vehiculo {
    public:
        Bicicleta(const std::string& i) ;
        virtual ~Bicicleta() ;
        virtual void mover() ;
        virtual void cambiar() ;
        virtual Bicicleta* clone() const ;
    protected:
        Bicicleta(const Bicicleta& o) ;
    private:
        //---- Atributos ----
        int plato ;
    };
}
#endif
```

```
#include "bicicleta.hpp"
namespace umalcc {
    Bicicleta::~Bicicleta() {}
    Bicicleta::Bicicleta(const std::string& i)
        : Vehiculo(i), plato(0) {}
    Bicicleta::Bicicleta(const Bicicleta& o)
        : Vehiculo(o), plato(o.plato) {}
    void Bicicleta::mover() {
        Vehiculo::mover(plato + 1) ;
    }
    void Bicicleta::cambiar() {
        plato = (plato + 1) % 3 ;
    }
    Bicicleta* Bicicleta::clone() const {
        return new Bicicleta(*this) ;
    }
}
```

Así, la clase polimórfica `Bicicleta` hereda tanto los atributos (`ident` y `posicion`) como los métodos (`id()`, `estacionar()`, `mover()` y `clone()`) de la clase base `vehiculo`, pero *redefine* los métodos `mover()` y `clone()` de la clase base, proporcionando una nueva implementación y comportamiento. Además, añade un nuevo atributo (`plato`) y un nuevo método (`cambiar()`) a la clase `Bicicleta`.

13.3. Utilización de Clases Polimórficas

Para que el *polimorfismo* y la *vinculación dinámica* sean efectivas, se debe trabajar con los objetos a través de *punteros*.

El *polimorfismo* permite que un puntero a un objeto de una *clase derivada* pueda ser considerado y utilizado como si fuera un puntero a un objeto de la *clase base*, y proporciona un soporte adecuado para el *Principio de sustitución*, mediante el cual, un objeto de la clase derivada puede sustituir a un objeto de la clase base, allí donde sea necesario. Por ejemplo:

```
Bicicleta* ptr_bicicleta = new Bicicleta("B123") ;
Vehiculo* ptr_vehiculo = ptr_bicicleta ;
```

Sin embargo, la dirección de correspondencia opuesta no se mantiene, ya que no todos los punteros a objetos de la clase base son también punteros a objetos de la clase derivada. Por ejemplo:

```
Automovil* ptr_automovil = ptr_vehiculo ;  // Error ptr_vehiculo apunta a un objeto Bicicleta
```

Para realizar la conversión opuesta, hay que utilizar una operación especial de *casting dinámico*, que comprueba en tiempo de ejecución si el objeto real puede ser convertido al tipo que se solicita:

```
Bicicleta* ptr_bicicleta = dynamic_cast<Bicicleta*>(ptr_vehiculo) ;
```

Si el tipo real del objeto puede ser convertido al tipo especificado en la conversión, entonces el operador `dynamic_cast<>()` produce un puntero al objeto del tipo especificado. Sin embargo, si la condición anterior no se cumple, y la conversión de tipos no es adecuada, entonces devuelve `NULL`.

```
Bicicleta* ptr_bicicleta = dynamic_cast<Bicicleta*>(ptr_vehiculo) ;
if (ptr_bicicleta != NULL) {
    ptr_bicicleta->cambiar();
}
```

La *vinculación dinámica* permite que las clases derivadas puedan *redefinir* el comportamiento de los métodos definidos en la clase base. Así, la *vinculación dinámica* permite, en *contextos*

polimórficos, que los métodos *virtuales* invocados se seleccionen adecuadamente, en tiempo de ejecución, dependiendo del *tipo real* del objeto, y no del *tipo aparente*. Por ejemplo, la siguiente invocación al método `mover()` a través de un puntero de tipo `Bicicleta*` ejecutará la definición proporcionada por la clase `Bicicleta`:

```
ptr_bicicleta->mover() ;
```

Pero la siguiente invocación al método `mover()` a través de un puntero de tipo `Vehiculo*` tambien ejecutará la definición proporcionada por la clase `Bicicleta`, ya que en realidad, `ptr_vehiculo` es un puntero a un objeto polimórfico creado como un objeto de tipo real `Bicicleta`:

```
ptr_vehiculo->mover() ;
```

Además, la invocación al destructor de un objeto, mediante el operador `delete`, también se realiza adecuadamente y se invoca al destructor real del objeto de tipo `Bicicleta`, aunque el puntero sea un puntero a una clase `Vehiculo`:

```
delete ptr_vehiculo ;
```

A continuación se muestra un utilización simple de la jerarquía de clases definida anteriormente.

```
#include <iostream>
#include "vehiculo.hpp"
#include "bicicleta.hpp"
#include "automovil.hpp"
using namespace std;
using namespace umalcc;
void proceso(Vehiculo* v)
{
    cout << "Estacionar " << v->id() << endl;
    v->estacionar() ;
}
int main()
{
    Bicicleta* bx = new Bicicleta("B001") ;
    proceso(bx) ;
    Vehiculo* b = bx ;
    Vehiculo* a = new Automovil("A002") ;
    proceso(a) ;
    a->mover() ;
    b->mover() ;
    bx->estacionar() ;
    bx->cambiar() ;
    bx->mover() ;
    // b->cambiar() ;    // No es posible
    // a->repostar(10) ; // No es posible
    Automovil* ax = dynamic_cast<Automovil*>(a) ;
    if (ax != NULL) {
        ax->estacionar() ;
        ax->repostar(10) ;
        ax->mover() ;
    }
    Vehiculo* v = a->clone() ;
    v->mover() ;
    delete v ;
    delete a ;
    delete b ;
}
```

Así, se pueden apreciar las siguientes características:

- El subprograma `proceso()` puede recibir como parámetro cualquier puntero a un objeto de la clase `Vehiculo` o de sus clases derivadas (`Bicicleta` y `Automovil`) gracias al *polimorfismo*.

- Las invocaciones a los métodos `id()`, `estacionar()` y `mover()` son adecuadas gracias a la *vinculación dinámica*, e invocarán a las implementaciones correspondientes dependiendo del tipo real del objeto, y no del tipo aparente del puntero.
- Las invocaciones a los métodos `id()` y `estacionar()` se pueden realizar tanto a través de punteros a `Vehiculo`, como a través de punteros a `Bicicleta` y `Automovil`, gracias a que estas clases han *heredado* dichos métodos de la clase base.
- La invocación al método `cambiar()` o `repostar()` sólo puede realizarse a través de punteros a la clase `Bicicleta` o `Automovil` (y derivadas), pero no a través de punteros a la clase `Vehiculo`, ya que esta clase no proporciona de tales métodos.
- Un puntero a un objeto de la clase `Bicicleta` o de la clase `Automovil` puede ser asignado a un puntero de la clase `Vehiculo` gracias al *polimorfismo*.
- El operador `dynamic_cast<>()` permite convertir un puntero a un objeto de la *clase base* a un puntero a un objeto de la *clase derivada si y solo si* el *tipo real* del objeto es de la clase derivada, o alguno de sus derivados.
- El método `clone()` realiza una duplicación y copia del objeto real gracias a la *vinculación dinámica*.
- La invocación a `delete` destruye cada objeto adecuadamente según su *tipo real* gracias a la *vinculación dinámica*.

13.4. Ejemplo

A continuación se muestra la definición de la clase *no-polmórfica* `Parking` que permite aparcar los vehículos de la jerarquía de clases definida anteriormente.

```
#ifndef parking_hpp_
#define parking_hpp_
#include "vehiculo.hpp"
#include <array>
#include <string>
namespace umalcc {
    class Parking {    // CLASE NO-POLIMORFICA
    public:
        ~Parking();
        Parking();
        Parking(const Parking& o);
        Parking& operator=(const Parking& o);
        void mostrar() const;
        void anyadir(Vehiculo* v, bool& ok);
        Vehiculo* extraer(const std::string& id);
    private:
        static const int MAX = 100;
        typedef std::array<Vehiculo*, MAX> Park;
        //--
        int buscar(const std::string& id) const;
        void destruir();
        void copiar(const Parking& o);
        //--
        int n_v;
        Park parking;
    };
}
#endif
```

A continuación se muestra la implementación de la clase *no-polmórfica* `Parking` que permite aparcar los vehículos de la jerarquía de clases definida anteriormente. En ella, se puede apreciar

como se invoca al método `clone()` cuando se va a duplicar un objeto de la clase `Parking`. Así mismo, también se puede apreciar como se invocan a los metodos `id()`, `estacionar()` y `mover()` según sea necesario.

```
#include "parking.hpp"
#include <iostream>
using namespace std;
namespace umalcc {
  Parking::~Parking() { destruir(); }
  Parking::Parking() : n_v(0), parking() {}
  Parking::Parking(const Parking& o)
      : n_v(), parking() { copiar(o); }
  Parking& Parking::operator=(const Parking& o) {
      if (this != &o) {
          destruir();
          copiar(o);
      }
      return *this;
  }
  void Parking::anyadir(Vehiculo* v, bool& ok) {
      ok = ((v != NULL)&&(n_v < MAX));
      if (ok) {
          v->estacionar();
          parking[n_v] = v;
          ++n_v;
      }
  }
  void Parking::destruir() {
      for (int i = 0; i < n_v; ++i) {
          delete parking[i];
      }
      n_v = 0;
  }
  void Parking::mostrar() const {
      for (int i = 0; i < n_v; ++i) {
          cout << parking[i]->id() << endl;
      }
  }
  Vehiculo* Parking::extraer(const std::string& id) {
      Vehiculo* v = NULL;
      int i = buscar(id);
      if (i < n_v) {
          v = parking[i];
          --n_v;
          parking[i] = parking[n_v];
          v->mover();
      }
      return v;
  }
  int Parking::buscar(const std::string& id) const {
      int i = 0;
      while ((i < n_v)&&(id != parking[i]->id())) {
          ++i;
      }
      return i;
  }
  void Parking::copiar(const Parking& o) {
      n_v = o.n_v;
      for (int i = 0; i < o.n_v; ++i) {
          parking[i] = o.parking[i]->clone();
      }
  }
}
```

Finalmente, el siguiente código muestra como se crean diferentes objetos de las clases definidas anteriomente, se almacenan en un aparcamiento, y se realizan diferentes manipulaciones de los objetos. Nótese como el destructor del `Parking` se encargara de destruir los objetos allí almacenados.

```
#include "vehiculo.hpp"
#include "bicicleta.hpp"
#include "automovil.hpp"
#include "parking.hpp"
#include <iostream>
using namespace std;
using namespace umalcc;
int main()
{
    bool ok;
    Parking park;
    Automovil* a1 = new Automovil("A1");
    Bicicleta* b1 = new Bicicleta("B1");
    Vehiculo*  a2 = new Automovil("A2");
    park.anyadir(a1, ok);
    park.anyadir(b1, ok);
    park.anyadir(a2, ok);
    Parking aux = park;
    Vehiculo*  v = aux.extraer("A1");
    if (v != NULL) {
        cout << "Vehiculo: " << v->id() << endl;
        v->mover();
    }
    aux.mostrar();
    Automovil* ax = dynamic_cast<Automovil*>(v);
    if (ax != NULL) {
        ax->repostar(20);
    }
    park.mostrar();
    delete v;
}
```



Capítulo 14

Introducción a los Contenedores de la Biblioteca Estándar (STL)

Los *contenedores* de la biblioteca estándar proporcionan un método general para almacenar y acceder a una colección de elementos homogéneos, proporcionando cada uno de ellos diferentes características que los hacen adecuados a diferentes necesidades.

En este capítulo introductorio se mostrarán las principales operaciones que se pueden realizar con los siguientes contenedores: el tipo `vector` y el tipo `deque` (para el tipo `array` véase 6.4). Así como con los siguientes *adaptadores de contenedores*: el tipo `stack` y el tipo `queue` de la biblioteca estándar, que implementan el *TAD Pila* y el *TAD Cola* respectivamente.

La biblioteca estándar también define otros tipos de contenedores optimizados para diferentes circunstancias, pero no serán explicados debido a que su estudio requiere mayores conocimientos que los obtenidos en un curso introductorio.

Contenedor	**Tipo**	**Acceso**	**Inserción**	**Eliminación**
`stack` (adaptador)	*TAD Pila*	Directo (al final)	Al final	Al final
`queue` (adaptador)	*TAD Cola*	Directo (al principio)	Al final	Al principio
`array`	Secuencia	Directo (pos)	–	–
`vector`	Secuencia	Directo (pos)	Al final	Al final
`deque`	Secuencia	Directo (pos)	Al final + al principio	Al final + Al principio
`list`	Secuencia	Secuencial (bidir)	Cualquier posición	Cualquier posición
`forward_list`	Secuencia	Secuencial (fw)	Cualquier posición	Cualquier posición
`map`	Asociativo	Binario por clave	Por Clave	Por Clave
`set`	Asociativo	Binario por clave	Por Clave	Por Clave
`multimap`	Asociativo	Binario por clave	Por Clave	Por Clave
`multiset`	Asociativo	Binario por clave	Por Clave	Por Clave
`unordered_map`	Asociativo	Hash por clave	Por Clave	Por Clave
`unordered_set`	Asociativo	Hash por clave	Por Clave	Por Clave
`unordered_multimap`	Asociativo	Hash por clave	Por Clave	Por Clave
`unordered_multiset`	Asociativo	Hash por clave	Por Clave	Por Clave

Paso de Parámetros de Contenedores

Los contenedores de la biblioteca estándar se pueden pasar como parámetros a subprogramas como cualquier otro tipo compuesto, y por lo tanto se aplican los mecanismos de paso de parámetros para tipos compuestos explicados en la sección 6.1. Es decir, los parámetros de entrada se pasarán por referencia constante, mientras que los parámetros de salida y entrada/salida se pasarán por referencia.

Así mismo, como norma general, salvo excepciones, no es adecuado que las funciones retornen valores de tipos de los contenedores, debido a la sobrecarga que generalmente conlleva dicha operación para el caso de los tipos compuestos. En estos casos suele ser más adecuado que el valor se devuelva como un parámetro por referencia.

14.1. Vector

El contenedor de tipo `vector<...>` representa una secuencia de elementos homogéneos optimizada para el acceso directo a los elementos según su posición, así como también para la inserción de elementos al final de la secuencia y para la eliminación de elementos del final de la secuencia. Para utilizar un contenedor de tipo `vector` se debe incluir la biblioteca estándar `<vector>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <vector>
```

El tipo `vector` es similar al tipo `array`, salvo en el hecho de que los vectores se caracterizan porque su tamaño puede crecer en tiempo de ejecución dependiendo de las necesidades surgidas durante la ejecución del programa. Por ello, a diferencia de los arrays, no es necesario especificar un tamaño fijo y predeterminado en tiempo de compilación respecto al número de elementos que pueda contener.

El número máximo de elementos que se pueden almacenar en una variable de tipo `vector` no está especificado, y se pueden almacenar elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Nótese que en los siguientes ejemplos, por simplicidad, tanto el número de elementos como el valor inicial de los mismos están especificados mediante valores constantes, sin embargo, también se pueden especificar como valores de variables y expresiones calculados en tiempo de ejecución.

Instanciación del Tipo Vector

Se pueden definir explícitamente instanciaciones del tipo `vector` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Vect_Int` como un tipo vector de números enteros.

```
typedef std::vector<int> Vect_Int ;
```

Las siguientes definiciones declaran el tipo `Matriz` como un vector de dos dimensiones de números enteros.

```
typedef std::vector<int>  Fila ;
typedef std::vector<Fila> Matriz ;
```

Construcción de un Objeto de Tipo Vector

Se pueden definir variables de un tipo vector previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`v1` y `v2`) de tipo vector de números enteros, así como la variable `m` de tipo vector de dos dimensiones de números enteros.

```
int main()
{
    Vect_Int v1 ;           // vector de enteros vacío
    std::vector<int> v2 ;   // vector de enteros vacío
    Matriz m ;              // vector de dos dimensiones de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `vector` crea un objeto `vector` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible crear un objeto vector con un número inicial de elementos con un valor inicial por defecto, al que posteriormente se le podrán añadir nuevos elementos. Este número inicial de elementos puede ser tanto una constante, como el valor de una variable calculado en tiempo de ejecución.

```
int main()
{
    Vect_Int v1(10) ;       // vector con 10 enteros con valor inicial sin definir
    Matriz m(10, Fila(5)) ; // matriz de 10x5 enteros con valor inicial sin definir
    // ...
}
```

Así mismo, también se puede especificar el valor que tomarán los elementos creados inicialmente.

```
int main()
{
    Vect_Int v1(10, 3) ;       // vector con 10 enteros con valor inicial 3
    Matriz m(10, Fila(5, 3)) ; // matriz de 10x5 enteros con valor inicial 3
    // ...
}
```

También es posible inicializar un vector con el contenido de otro vector de igual tipo, invocando al constructor de copia:

```
int main()
{
    Vect_Int v1(10, 3) ;     // vector con 10 enteros con valor inicial 3
    Vect_Int v2(v1) ;        // vector con el mismo contenido de v1
    Vect_Int v3 = v1 ;       // vector con el mismo contenido de v1
    Vect_Int v4 = Vect_Int(7, 5) ; // vector con 7 elementos de valor 5
    // ...
}
```

Asignación de un Objeto de Tipo Vector

Es posible la asignación de vectores de igual tipo. En este caso, se destruye el valor anterior del vector destino de la asignación.

```
int main()
{
    Vect_Int v1(10, 3) ;    // vector con 10 enteros con valor inicial 3
    Vect_Int v2 ;           // vector de enteros vacío

    v2 = v1 ;               // asigna el contenido de v1 a v2
    v2.assign(5, 7) ;       // asigna 5 enteros con valor inicial 7
    v2 = Vect_Int(5, 7) ;   // asigna un vector con 5 elementos de valor 7
}
```

Así mismo, también es posible intercambiar (*swap* en inglés) de forma eficiente el contenido entre dos vectores utilizando el método `swap`. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 5) ;    // v1 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
    Vect_Int v2(5, 7) ;     // v2 = { 7, 7, 7, 7, 7 }

    v1.swap(v2) ;           // v1 = { 7, 7, 7, 7, 7 }
                            // v2 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
}
```

Control sobre los Elementos de un Vector

El número de elementos actualmente almacenados en un vector se obtiene mediante el método `size()`. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 3) ;    // vector con 10 enteros con valor inicial 3
    int n = v1.size() ;     // número de elementos de v1
}
```

Es posible tanto añadir un elemento al final de un vector mediante el método `push_back(...)`, como eliminar el último elemento del vector mediante el método `pop_back()` (en este caso el vector no debe estar vacío). Así mismo, el método `clear()` elimina todos los elementos del vector. Por ejemplo:

```
int main()
{
    Vect_Int v(5, 0) ;                          // v = { 0, 0, 0, 0, 0 }
    for (int i = 1 ; i <= 3 ; ++i) {
        v.push_back(i) ;
    }                                           // v = { 0, 0, 0, 0, 0, 1, 2, 3 }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }                                           // muestra: 0 0 0 0 0 1 2 3
    cout << endl ;
    while (v.size() > 3) {
        v.pop_back() ;
    }                                           // v = { 0, 0, 0 }
    v.clear() ;                                 // v = { }
}
```

También es posible cambiar el tamaño del número de elementos almacenados en el vector. Así, el método `resize(...)` reajusta el número de elementos contenidos en un vector. Si el número especificado es menor que el número actual de elementos, se eliminarán del final del vector tantos elementos como sea necesario para reducir el vector hasta el número de elementos especificado. Si por el contrario, el número especificado es mayor que el número actual de elementos, entonces se añadirán al final del vector tantos elementos como sea necesario para alcanzar el nuevo número de elementos especificado (con el valor especificado o con el valor por defecto). Por ejemplo:

```
int main()
{
    Vect_Int v(10, 1) ;  // v = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
    v.resize(5) ;        // v = { 1, 1, 1, 1, 1 }
    v.resize(9, 2) ;     // v = { 1, 1, 1, 1, 1, 2, 2, 2, 2 }
    v.resize(7, 3) ;     // v = { 1, 1, 1, 1, 1, 2, 2 }
    v.resize(10) ;       // v = { 1, 1, 1, 1, 1, 2, 2, 0, 0, 0 }
}
```

Acceso a los Elementos de un Vector

Es posible acceder a cada elemento del vector individualmente, según el índice de la posición que ocupe, tanto para obtener su valor almacenado, como para modificarlo mediante el operador de indexación `[]`. El primer elemento ocupa la posición cero (0), y el último elemento almacenado en el vector `v` ocupa la posición `v.size()-1`. Por ejemplo:

```
int main()
{
    Vect_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v[i] = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }
```

```
    cout << endl ;
}
```

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un vector sean correctos y se encuentren dentro de los límites válidos del vector, por lo que será responsabilidad del programador comprobar que así sea.

Sin embargo, en *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

También es posible acceder a un determinado elemento mediante el método `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range` que abortará la ejecución del programa. Se puede tanto utilizar como modificar el valor de este elemento.

```
int main()
{
    Vect_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v.at(i) = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v.at(i) << " " ;
    }
    cout << endl ;
}
```

Comparación Lexicográfica entre Vectores

Es posible realizar la comparación lexicográfica (==, !=, >, >=, <, <=) entre vectores del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes del vector. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 7) ;  // v1 = { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 }
    Vect_Int v2(5, 3) ;   // v2 = { 3, 3, 3, 3, 3 }
    if (v1 == v2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (v1 < v2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.2. Deque

El contenedor de tipo `deque<...>` representa una secuencia de elementos homogéneos optimizada para el acceso directo a los elementos según su posición, así como también para la inserción de elementos al principio y al final de la secuencia y para la eliminación de elementos del principio y del final de la secuencia. Para utilizar un contenedor de tipo `deque` se debe incluir la biblioteca estándar `<deque>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <deque>
```

El contenedor deque presenta el mismo interfaz público que el contenedor vector, pero añade dos métodos nuevos para facilitar la inserción y eliminación de elementos al principio de la secuencia (push_front(...) y pop_front()).

El número máximo de elementos que se pueden almacenar en una variable de tipo deque no está especificado, y se pueden almacenar elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Deque

Se pueden definir explícitamente instanciaciones del tipo deque para tipos de elementos concretos mediante la declaración typedef. Por ejemplo la siguiente definición declara el tipo Deque_Int como un tipo deque de números enteros.

```
typedef std::deque<int> Deque_Int ;
```

Construcción de un Objeto de Tipo Deque

Se pueden definir variables de un tipo deque previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (v1 y v2) de tipo deque de números enteros.

```
int main()
{
    Deque_Int v1 ;          // deque de enteros vacío
    std::deque<int> v2 ;    // deque de enteros vacío
    // ...
}
```

El constructor por defecto del tipo deque crea un objeto deque inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible crear un objeto deque con un número inicial de elementos con un valor inicial por defecto, al que posteriormente se le podrán añadir nuevos elementos. Este número inicial de elementos puede ser tanto una constante, como el valor de una variable calculado en tiempo de ejecución.

```
int main()
{
    Deque_Int v1(10) ;       // deque con 10 enteros con valor inicial sin definir
    // ...
}
```

Así mismo, también se puede especificar el valor que tomarán los elementos creados inicialmente.

```
int main()
{
    Deque_Int v1(10, 3) ;       // deque con 10 enteros con valor inicial 3
    // ...
}
```

También es posible inicializar un deque con el contenido de otro deque de igual tipo, invocando al constructor de copia:

```
int main()
{
    Deque_Int v1(10, 3) ;     // deque con 10 enteros con valor inicial 3
    Deque_Int v2(v1) ;        // deque con el mismo contenido de v1
    Deque_Int v3 = v1 ;       // deque con el mismo contenido de v1
    Deque_Int v4 = Deque_Int(7, 5) ; // deque con 7 elementos de valor 5
    // ...
}
```

Asignación de un Objeto de Tipo Deque

Es posible la asignación de deques de igual tipo. En este caso, se destruye el valor anterior del deque destino de la asignación.

```
int main()
{
    Deque_Int v1(10, 3) ;    // deque con 10 enteros con valor inicial 3
    Deque_Int v2 ;           // deque de enteros vacío

    v2 = v1 ;                // asigna el contenido de v1 a v2
    v2.assign(5, 7) ;        // asigna 5 enteros con valor inicial 7
    v2 = Deque_Int(5, 7) ;   // asigna un deque con 5 elementos de valor 7
}
```

Así mismo, también es posible intercambiar (*swap* en inglés) de forma eficiente el contenido entre dos deques utilizando el método `swap`. Por ejemplo:

```
int main()
{
    Deque_Int v1(10, 5) ;    // v1 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
    Deque_Int v2(5, 7) ;     // v2 = { 7, 7, 7, 7, 7 }

    v1.swap(v2) ;            // v1 = { 7, 7, 7, 7, 7 }
                             // v2 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
}
```

Control sobre los Elementos de un Deque

El número de elementos actualmente almacenados en un deque se obtiene mediante el método `size()`. Por ejemplo:

```
int main()
{
    Deque_Int v1(10, 3) ;    // deque con 10 enteros con valor inicial 3
    int n = v1.size() ;      // número de elementos de v1
}
```

Es posible tanto añadir un elemento al final de un deque mediante el método `push_back(...)`, como eliminar el último elemento del deque mediante el método `pop_back()` (en este caso el deque no debe estar vacío). Así mismo, el método `clear()` elimina todos los elementos del deque. Por ejemplo:

```
int main()
{
    Deque_Int v(5, 0) ;                          // v = { 0, 0, 0, 0, 0 }
    for (int i = 1 ; i <= 3 ; ++i) {
        v.push_back(i) ;
    }                                            // v = { 0, 0, 0, 0, 0, 1, 2, 3 }
    for (int i = 1 ; i <= 2 ; ++i) {
        v.push_front(i) ;
    }                                            // v = { 2, 1, 0, 0, 0, 0, 0, 1, 2, 3 }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }                                            // muestra: 2 1 0 0 0 0 0 1 2 3
    cout << endl ;
    while (v.size() > 5) {
        v.pop_back() ;
    }                                            // v = { 2, 1, 0, 0, 0 }
    while (v.size() > 3) {
        v.pop_front() ;
```

```
    }                                       // v = { 0, 0, 0 }
    v.clear() ;                             // v = { }
}
```

También es posible cambiar el tamaño del número de elementos almacenados en el deque. Así, el método `resize(...)` reajusta el número de elementos contenidos en un deque. Si el número especificado es menor que el número actual de elementos, se eliminarán del final del deque tantos elementos como sea necesario para reducir el deque hasta el número de elementos especificado. Si por el contrario, el número especificado es mayor que el número actual de elementos, entonces se añadirán al final del deque tantos elementos como sea necesario para alcanzar el nuevo número de elementos especificado (con el valor especificado o con el valor por defecto). Por ejemplo:

```
int main()
{
    Deque_Int v(10, 1) ;  // v = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
    v.resize(5) ;         // v = { 1, 1, 1, 1, 1 }
    v.resize(9, 2) ;      // v = { 1, 1, 1, 1, 1, 2, 2, 2, 2 }
    v.resize(7, 3) ;      // v = { 1, 1, 1, 1, 1, 2, 2 }
    v.resize(10) ;        // v = { 1, 1, 1, 1, 1, 2, 2, 0, 0, 0 }
}
```

Acceso a los Elementos de un Deque

Es posible acceder a cada elemento del deque individualmente, según el índice de la posición que ocupe, tanto para obtener su valor almacenado, como para modificarlo mediante el operador de indexación `[]`. El primer elemento ocupa la posición cero (0), y el último elemento almacenado en el deque `v` ocupa la posición `v.size()-1`. Por ejemplo:

```
int main()
{
    Deque_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v[i] = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }
    cout << endl ;
}
```

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un deque sean correctos y se encuentren dentro de los límites válidos del deque, por lo que será responsabilidad del programador comprobar que así sea.

Sin embargo, en *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

También es posible acceder a un determinado elemento mediante el método `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range` que abortará la ejecución del programa. Se puede tanto utilizar como modificar el valor de este elemento.

```
int main()
{
    Deque_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v.at(i) = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v.at(i) << " " ;
    }
    cout << endl ;
}
```

Comparación Lexicográfica entre Deques

Es posible realizar la comparación lexicográfica (==, !=, >, >=, <, <=) entre deques del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes del deque. Por ejemplo:

```
int main()
{
    Deque_Int v1(10, 7) ;  // v1 = { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 }
    Deque_Int v2(5, 3) ;   // v2 = { 3, 3, 3, 3, 3 }
    if (v1 == v2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (v1 < v2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.3. Stack

El adaptador de contenedor de tipo `stack<...>` representa el tipo abstracto de datos *Pila*, como una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en orden inverso al orden de inserción), de tal forma que el primer elemento que sale de la pila es el último elemento que ha sido introducido en ella. Además, también es posible comprobar si la pila contiene elementos, de tal forma que no se podrá sacar ningún elemento de una pila vacía. Para utilizar un adaptador de contenedor de tipo `stack` se debe incluir la biblioteca estándar `<stack>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <stack>
```

El número máximo de elementos que se pueden almacenar en una variable de tipo `stack` no está especificado, y se pueden introducir elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Stack

Se pueden definir explícitamente instanciaciones del tipo `stack` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Stack_Int` como un tipo pila de números enteros.

```
typedef std::stack<int> Stack_Int ;
```

Construcción de un Objeto de Tipo Pila

Se pueden definir variables de un tipo pila previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`s1` y `s2`) de tipo pila de números enteros.

```
int main()
{
    Stack_Int s1 ;        // stack de enteros vacío
    std::stack<int> s2 ;  // stack de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `stack` crea un objeto `stack` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible inicializar una pila con el contenido de otra pila de igual tipo:

```
int main()
{
    Stack_Int s1 ;       // stack de enteros vacío
    // ...
    Stack_Int s2(s1) ;   // stack con el mismo contenido de s1
    Stack_Int s3 = s1 ;  // stack con el mismo contenido de s1
    Stack_Int s4 = Stack_Int() ;  // copia el contenido de stack vacío
    // ...
}
```

Asignación de un Objeto de Tipo Pila

Es posible la asignación de pilas de igual tipo. En este caso, se destruye el valor anterior de la pila destino de la asignación.

```
int main()
{
    Stack_Int s1 ;       // stack de enteros vacío
    Stack_Int s2 ;       // stack de enteros vacío

    s2 = s1 ;            // asigna el contenido de s1 a s2
    s2 = Stack_Int() ;   // asigna el contenido de stack vacío
}
```

Control y Acceso a los Elementos de una Pila

Es posible tanto añadir un elemento una pila mediante el método `push(...)`, como eliminar el último elemento introducido en la pila mediante el método `pop()` (en este caso la pila no debe estar vacía).

Por otra parte, el método `empty()` indica si una pila está vacía o no, mientras que el número de elementos actualmente almacenados en una pila se obtiene mediante el método `size()`.

Así mismo, se puede acceder al último elemento introducido en la pila mediante el método `top()`. Se puede tanto utilizar como modificar el valor de este elemento (en este caso la pila no debe estar vacía).

Por ejemplo:

```
int main()
{
    Stack_Int s ;                          // s = { }
    for (int i = 1 ; i <= 3 ; ++i) {
        s.push(i) ;
    }                                     // s = { 1, 2, 3 }

    s.top() = 5 ;                          // s = { 1, 2, 5 }

    s.pop() ;                              // s = { 1, 2 }
    s.pop() ;                              // s = { 1 }
    s.push(7) ;                            // s = { 1, 7 }
    s.push(9) ;                            // s = { 1, 7, 9 }

    cout << s.size() << endl ;             // muestra: 3

    while (! s.empty()) {
        cout << s.top() << " " ;           // muestra: 9 7 1
        s.pop() ;
```

```
    }                                          // s = { }
    cout << endl ;
}
```

Comparación Lexicográfica entre Pilas

Es posible realizar la comparación lexicográfica (==, !=, >, >=, <, <=) entre pilas del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes de la pila. Por ejemplo:

```
int main()
{
    Stack_Int s1 ;
    Stack_Int s2 ;
    // ...
    if (s1 == s2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (s1 < s2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.4. Queue

El adaptador de contenedor de tipo `queue<...>` representa el tipo abstracto de datos *Cola*, como una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en el mismo orden al orden de inserción), de tal forma que el primer elemento que sale de la cola es el primer elemento que ha sido introducido en ella. Además, también es posible comprobar si la cola contiene elementos, de tal forma que no se podrá sacar ningún elemento de una cola vacía. Para utilizar un adaptador de contenedor de tipo `queue` se debe incluir la biblioteca estándar `<queue>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <queue>
```

El número máximo de elementos que se pueden almacenar en una variable de tipo `queue` no está especificado, y se pueden introducir elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Queue

Se pueden definir explícitamente instanciaciones del tipo `queue` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Queue_Int` como un tipo cola de números enteros.

```
typedef std::queue<int> Queue_Int ;
```

Construcción de un Objeto de Tipo Cola

Se pueden definir variables de un tipo cola previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`c1` y `c2`) de tipo cola de números enteros.

```
int main()
{
    Queue_Int c1 ;        // queue de enteros vacío
    std::queue<int> c2 ;  // queue de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `queue` crea un objeto `queue` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible inicializar una cola con el contenido de otra cola de igual tipo:

```
int main()
{
    Queue_Int c1 ;       // queue de enteros vacío
    // ...
    Queue_Int c2(c1) ;   // queue con el mismo contenido de c1
    Queue_Int c3 = c1 ;  // queue con el mismo contenido de c1
    Queue_Int c4 = Stack_Int() ;  // copia el contenido de queue vacío
    // ...
}
```

Asignación de un Objeto de Tipo Cola

Es posible la asignación de colas de igual tipo. En este caso, se destruye el valor anterior de la cola destino de la asignación.

```
int main()
{
    Queue_Int c1 ;       // queue de enteros vacío
    Queue_Int c2 ;       // queue de enteros vacío

    c2 = c1 ;            // asigna el contenido de c1 a c2
    c2 = Queue_Int() ;   // asigna el contenido de queue vacío
}
```

Control y Acceso a los Elementos de una Cola

Es posible tanto añadir un elemento una cola mediante el método `push(...)`, como eliminar el primer elemento introducido en la cola mediante el método `pop()` (en este caso la cola no debe estar vacía).

Por otra parte, el método `empty()` indica si una cola está vacía o no, mientras que el número de elementos actualmente almacenados en una cola se obtiene mediante el método `size()`.

Así mismo, se puede acceder al último elemento introducido en la cola mediante el método `back()`, así como al primer elemento introducido en ella mediante el método `front()`. Se pueden tanto utilizar como modificar el valor de estos elementos (en este caso la cola no debe estar vacía).

Por ejemplo:

```
int main()
{
    Queue_Int c ;                            // c = { }
    for (int i = 1 ; i <= 3 ; ++i) {
        c.push(i) ;
    }                                       // c = { 1, 2, 3 }

    c.front() = 6 ;                          // c = { 6, 2, 3 }
    c.back() = 5 ;                           // c = { 6, 2, 5 }

    c.pop() ;                                // c = { 2, 5 }
    c.pop() ;                                // c = { 5 }
```

```
    c.push(7) ;                          // c = { 5, 7 }
    c.push(9) ;                          // c = { 5, 7, 9 }

    cout << c.size() << endl ;           // muestra: 3

    while (! c.empty()) {
        cout << c.front() << " " ;       // muestra: 5 7 9
        c.pop() ;
    }                                    // c = { }
    cout << endl ;
}
```

Comparación Lexicográfica entre Colas

Es posible realizar la comparación lexicográfica (==, !=, >, >=, <, <=) entre colas del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes de la cola. Por ejemplo:

```
int main()
{
    Queue_Int c1 ;
    Queue_Int c2 ;
    // ...
    if (c1 == c2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (c1 < c2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.5. Resolución de Problemas Utilizando Contenedores

Ejemplo 1: Agentes de Ventas

Diseñe un programa que lea y almacene las ventas realizadas por unos *agentes de ventas*, de tal forma que se eliminen aquellos agentes cuyas ventas sean inferiores a la media de las ventas realizadas.

```
//------------------------------------------------------------------
#include <iostream>
#include <vector>
#include <string>
using namespace std ;

struct Agente {
    string nombre ;
    double ventas ;
} ;

typedef vector<Agente> VAgentes ;

void leer (VAgentes& v)
{
    v.clear() ;
```

```
    Agente a ;
    cout << "Introduzca Nombre: " ;
    getline(cin, a.nombre) ;
    while (( ! cin.fail()) && (a.nombre.size() > 0)) {
        cout << "Introduzca Ventas: " ;
        cin >> a.ventas ;
        cin.ignore(1000, '\n') ;
        v.push_back(a) ;
        cout << "Introduzca Nombre: " ;
        getline(cin, a.nombre) ;
    }
}

double media(const VAgentes& v)
{
    double suma=0.0 ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        suma += v[i].ventas ;
    }
    return suma/double(v.size()) ;
}

void purgar(VAgentes& v, double media)
{
    // altera el orden secuencial de los elementos
    int i = 0 ;
    while (i < int(v.size())) {
        if (v[i].ventas < media) {
            v[i] = v[v.size()-1] ;
            v.pop_back() ;
        } else {
            ++i ;
        }
    }
}

void purgar_ordenado(VAgentes& v, double media)
{
    // mantiene el orden secuencial de los elementos
    int k = 0 ;
    while ((k < int(v.size()))&&(v[k].ventas >= media)) {
        ++k;
    }
    for (int i = k ; i < int(v.size()) ; ++i) {
        if(v[i].ventas >= media) {
            v[k] = v[i] ;
            ++k ;
        }
    }
    v.resize(k) ;
}

void imprimir(const VAgentes& v)
{
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i].nombre << " " << v[i].ventas << endl ;
    }
}
```

```
int main ()
{
    VAgentes v ;
    leer(v) ;
    purgar(v, media(v)) ;
    imprimir(v) ;
}
//------------------------------------------------------------------
```

Ejemplo 2: Multiplicación de Matrices

Diseñe un programa que lea dos matrices de tamaños arbitrarios y muestre el resultado de multiplicar ambas matrices.

```
//------------------------------------------------------------------
#include <vector>
#include <iostream>
#include <iomanip>
using namespace std ;

typedef vector <double> Fila ;
typedef vector <Fila> Matriz ;

void imprimir(const Matriz& m)
{
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        for (int c = 0 ; c < int(m[f].size()) ; ++c) {
            cout << setw(10) << setprecision(4)
                 << m[f][c] << " " ;
        }
        cout << endl ;
    }
}

void leer(Matriz& m)
{
    int nf, nc ;
    cout << "Introduzca el numero de filas: " ;
    cin >> nf ;
    cout << "Introduzca el numero de columnas: " ;
    cin >> nc ;
    m = Matriz(nf, Fila (nc)) ; // copia de la matriz completa
    cout << "Introduzca los elementos: " << endl ;
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        for (int c = 0 ; c < int(m[f].size()) ; ++c) {
            cin >> m[f][c] ;
        }
    }
}

// otra opción más eficiente para la lectura de vectores
void leer_2(Matriz& m)
{
    int nf, nc ;
    cout << "Introduzca el numero de filas: " ;
    cin >> nf ;
    cout << "Introduzca el numero de columnas: " ;
    cin >> nc ;
    Matriz aux(nf, Fila (nc)) ;
    cout << "Introduzca los elementos: " << endl ;
```

```
    for (int f = 0 ; f < int(aux.size()) ; ++f) {
        for (int c = 0 ; c < int(aux[f].size()) ; ++c) {
            cin >> aux[f][c] ;
        }
    }
    m.swap(aux) ;  // evita la copia de la matriz completa
}

void multiplicar(const Matriz& m1, const Matriz& m2, Matriz& m3)
{
    m3.clear() ;
    if ((m1.size() > 0) && (m2.size() > 0) && (m2[0].size() > 0)
        && (m1[0].size() == m2.size())){
        Matriz aux(m1.size(), Fila(m2[0].size())) ;
        for (int f = 0 ; f < int(aux.size()) ; ++f) {
            for (int c = 0 ; c < int(aux[f].size()) ; ++c) {
                double suma = 0.0 ;
                for (int k = 0 ; k < int(m2.size()) ; ++k) {
                    suma += m1[f][k] * m2[k][c] ;
                }
                aux[f][c] = suma ;
            }
        }
        m3.swap(aux) ;  // evita la copia de la matriz completa
    }
}

int main()
{
    Matriz m1, m2, m3 ;
    leer(m1) ;
    leer(m2) ;
    multiplicar(m1, m2, m3) ;
    if (m3.size() == 0) {
        cout << "Error en la multiplicación de Matrices" << endl ;
    } else {
        imprimir(m3) ;
    }
}
//------------------------------------------------------------------
```

Capítulo 15

Bibliografía

- El Lenguaje de Programación C. 2.Ed.
  B.Kernighan, D. Ritchie
  Prentice Hall 1991

- The C++ Programming Language. Special Edition
  B. Stroustrup
  Addison Wesley 2000

Índice alfabético